Appunti di
LABORATORIO D'INFORMATICA

Prof. Bernardinello, Luca
a.a. 2010/2011
4 CFU

PAOLO DI GIGLIO

Ultima revisione:
10 maggio 2012

INDICE

PREFAZIONE

Ecco una raccolta di tutti gli appunti che ho preso durante il corso di Laboratorio d'Informatica tenuto dal prof. Luca Bernardinello, con qualche aggiunta e rivisitazione.

Sono sicuro che risulterà imprecisa e lacunosa in molte parti, pertanto invito tutti quanti leggano questo documento a segnalare errori, imprecisioni, carenze e quant'altro a `seek.and.destroy@live.it`. Qualora qualcuno avesse a disposizione del materiale aggiuntivo sarà il benvenuto.

Spero che qualcuno possa trarre beneficio dal mio lavoro.

Paolo Di Giglio.

1 LEZIONE I

24/03/2011

1.1 INTRODUZIONE INFORMALE ALL'INFORMATICA

L'*informatica* è una scienza che permette di risolvere per via algoritmica problemi come i seguenti.[1]

Si stabilisca quale percorso deve seguire un aereo per arrivare da un aeroporto ad un altro. Qualora ci fossero più percorsi, si trovi quello più breve. Tale problema si può risolvere schematizzando i percorsi e gli aeroporti. Può tornare utile rappresentare gli aeroporti per mezzo di punti e le eventuali rotte che li collegano con dei segmenti. Ora, non è più rilevante la struttura geografica dei luoghi o il percorso effettivamente seguito dal mezzo. Basta conoscere gli aeroporti, le rotte di collegamento e il loro rispettivo "peso" (ossia la distanza). Una rappresentazione come quella qui descritta si chiama *grafo* (vedi il paragrafo 6.1.1 a pagina 23);

Si è qui operato un processo di *astrazione*. Esso consiste nell'estrapolare dal problema reale tutti e soli i dati che servano a risolverlo, trascurando dettagli inutili e semplificando così la questione.

Un ladro entra in un magazzino. Ha a disposizione solo uno zaino che riesce a sostenere un peso finito con cui trasportare la refurtiva. Nel magazzino sono presenti oggetti di diverso valore e diverso peso. Il suo scopo è di massimizzare il suo guadagno cercando di portare via oggetti del maggior valore possibile. Tale questione si può esprimere in modo più formale come segue. Sia dato uno zaino che possa sopportare un determinato peso W. Siano dati inoltre N oggetti, ognuno dei quali caratterizzato da un peso w_i e un valore c_i con $i \in \{1, \ldots, N\}$. Si scelgano quali di questi oggetti mettere nello zaino per ottenere il maggiore valore senza eccedere nel peso sostenibile dallo zaino stesso [4].

Una soluzione possibile consiste nello scegliere gli oggetti da portare via in base al loro valore unitario, cioè al rapporto:

$$VU_i = \frac{c_i}{w_i}.$$

Ovviamente, un oggetto avrà priorità tanto maggiore quanto più grande sarà il suo valore unitario.

Si hanno 50 schedine numerate. Esse sono indicizzate dei numeri non progressivi compresi in un range che va da 1 a 10^6. Bisogna ridisporre tale schedine in ordine crescente. In particolare, ci sono dei vincoli. Supponiamo di avere a disposizione uno spazio che non permetta di vedere i numeri di tutte le schedine contemporaneamente. C'è, però, a abbastanza spazio per appoggiare contemporaneamente due mazzi.

Un procedimento risolutivo potrebbe essere il seguente. Si estrae una schedina s_1 dal mazzo non ordinato e la si appoggia accanto al mazzo stesso, in modo che il suo numero sia visibile. Se n'estrae una seconda s_2 e la si confronta con la prima. Se $s_1 > s_2$, si pone s_2 sopra s_1. Altrimenti si ripone s_2 in fondo al mazzo da cui è stata estratta. Si ripete tale procedimento fino a quando il primo mazzo non finisce. In informatica, tale metodo risolutivo prende talvolta il nome di *bubble sort*.

1 Possiamo assumere, per il momento che l'algoritmo sia un processo meccanico che produca dei risultati a partire da dei dati seguendo dei passaggi prestabiliti.

Tabella 1.1: Costo computazionale di un algoritmo bubble sort ($n \in \mathbb{N} \wedge n > 0$).

n. di carte	*n. di confronti*
50	49
49	48
48	47
⋮	⋮
n	$n-1$

1.1.1 Efficienza di un algoritmo

Quando si stende un algoritmo, bisogna cercare di renderlo il più efficiente e performante possibile. L'efficienza di un algoritmo, come ci si potrebbe aspettare, è inversamente proporzionale al numero di "calcoli" che l'esecutore deve compiere per giungere al risultato.

bubble sort

La tabella 1.1 mostra il numero massimo di confronti da compiere (in riferimento alla terza tipologia di problema) tramite un algoritmo di bubble sort in relazione al numero di carte che rimangono nel primo mazzo. In generale, supponendo di avere un mazzo non ordinato di n carte, con $n \in \mathbb{N}$, si hanno $(n-1)+(n-2)+\cdots+2+1$ confronti c, ossia:

$$c_{bs}(n) = \sum_{i=1}^{n-1} i = \frac{(n-2)(n-1)}{2}$$

dove, l'ultima uguaglianza è giustificata dall'algoritmo di Gauss per la somma dei primi n numeri naturali. Pertanto, per $n \to +\infty$, si ha che:

$$c_{bs}(n)_{n\to+\infty} = \sum_{i=1}^{n-1} i = \frac{(n-2)(n-1)}{2} \sim \frac{n^2}{2} \asymp n^2.$$

Raddoppiando il numero di carte, dunque, il numero di confronti quadruplica.

merge sort

Esistono molti altri metodi d'ordinamento, tra cui l'algoritmo *merge sort*[2]. Esso si basa un po' sull'antico detto romano "divide et impera" [3].

Come mostrato in figura 1.1 a fronte, si dividono i dati di partenza (nel nostro caso, le 50 schedine) per 2 fino ad ottenerne mazzi da due o una. Se il numero di dati non è divisibile per due, non importa; in questo caso si avrà almeno un mazzo da una schedina. Dopo aver suddiviso le schedine, si ordinano progressivamente partendo "dal basso".

Per quanto riguarda il costo di tale algoritmo, si devono effettuare all'incirca $\log_2 n$ suddivisioni. In ogni passaggio a ritroso, inoltre, bisogna compiere n confronti. Il costo dell'algoritmo sarà, quindi, $c_{ms}(n) \sim n \log_2 n$. Per $n \to +\infty$, si ha che:

$$c_{bs}(n)_{n\to+\infty} \sim n^2 \gg n \log_2 n \sim c_{ms}(n)_{n\to+\infty}.$$

Si riscontra, quindi, che l'algoritmo merge sort è più efficiente dell'algoritmo bubble sort.

1.2 PROGRAMMA

Il *programma* è un oggetto che si può presentare sotto varie forme. Le più importanti sono tre:

2 Inventato da John von Neumann nel 1945.

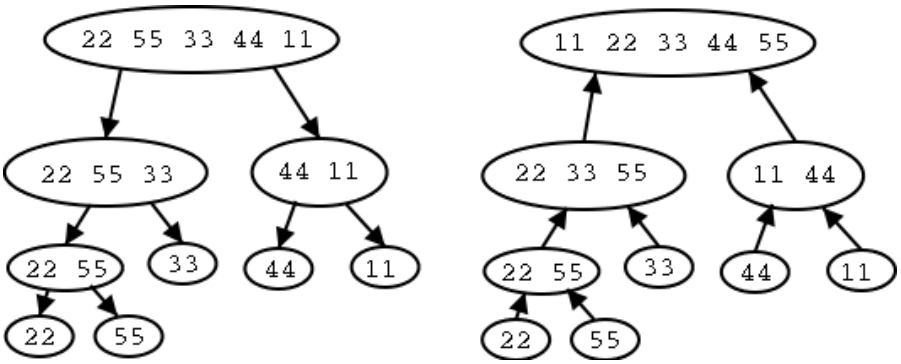


Figura 1.1: Rappresentazione dell'algoritmo merge sort.

IN CORSO D'ESECUZIONE. In questo caso, prende il nome di *processo*;

FILE ESEGUIBILE. Esso è essenzialmente un file (con estensione .exe in Windows) contenente un insieme d'istruzioni comprensibili dall'unità di calcolo. Una volta eseguito, si traduce in un processo. Il linguaggio in cui è scritto tale file prende il nome di *assembly* o *linguaggio macchina*. Quest'ultimo nome lascia intuire che tale file non è fatto per essere letto ed interpretato dall'uomo;

CODICE SORGENTE. Questo è un file di testo contenente istruzioni comprensibili anche all'uomo. È scritto in un *linguaggio di programmazione* (vedi il paragrafo 12.2.6 a pagina 55). Per ottenere un file eseguibile dal codice sorgente, c'è bisogno di "tradurre" quest'ultimo tramite un processo che, per ragioni storiche, prende il nome di *compilazione* (per maggiori dettagli, si rimanda a degli appunti sul sito del prof. Bernardinello).

2 LEZIONE II

31/03/2011

2.1 STRUTTURA GENERALE DI UN CALCOLATORE

Struttura di John von Neumann

La struttura di un calcolatore descritta di seguito è rimasta più o meno immutata nel corso del tempo, sin dagli anni '40. Essa è stata attribuita al matematico e informatico *John von Neumann* (figura 2.1a nella pagina seguente). Il calcolatore (figura 2.1b nella pagina successiva) è essenzialmente composto da:

CPU (*Central Process Unit*, cioè *Unità Centrale d'Elaborazione*), dove vengono trasformati e manipolati i dati;

RAM (*Random Access Memory*, ossia *Memoria di Lavoro*) è una memoria volatile. Nel caso di spegnimento del computer, i dati contenuti in essa vanno perduti. Serve per conservare i dati durante l'esecuzione di un programma (o l'elaborazione degli stessi);

I/O (unità di *Input/Output*, cioè d'ingresso e uscita) sono tutte quelle unità che consentono di immettere/acquisire dati nel/dal calcolatore. Sono, ad esempio: tastiera (input), mouse (input), monitor (output), stampante (output)...

MEMORIA PERMANENTE ossia quei supporti che conservano i dati a lungo termine, anche dopo lo spegnimento del calcolatore. Sono, ad esempio, dischi fissi, floppy, Compact Disc...

Le prime tre componenti elencate sono assolutamente necessarie ai fini del funzionamento del calcolatore. Esse comunicano tra di loro per mezzo del *bus*. Affinché le varie parti lavorino in modo sincrono, inoltre, in ogni macchina è presente anche un apparecchio simile ad un metronomo (il *clock*), che manda dei segnali ad ogni parte del computer. Il clock, quindi, stabilisce la frequenza di lavoro.

La CPU contiene dei registri. In ognuno di essi si possono conservare dei dati, che la CPU è in grado di leggere ed utilizzare. La RAM, in modo analogo, è organizzata in celle indicizzate. Ogni cella è individuata da un numero intero non negativo. Tali numeri sono progressivi e vanno da 0 a n. Affinché i programmi vengano eseguiti, è necessario memorizzare informazioni nella memoria di lavoro. Queste sono composte, in genere, sia dai dati da elaborare, sia dalle istruzioni tramite cui avverrà l'elaborazione.

Funzionamento della CPU

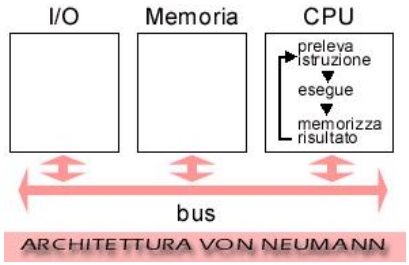
La macchina (più strettamente parlando, la CPU) per funzionare esegue tre operazioni principali:

FETCH; reperisce nella memoria di lavoro la prossima istruzione da eseguire. Tra i registri della CPU, infatti, ve n'è uno dedicato a contenere il numero della cella (chiamato anche *indirizzo*) della RAM contenente l'istruzione successiva;

DECODE; una volta trovata l'istruzione da eseguire è necessario che questa venga decodificata dalla CPU;

EXECUTE; avvenuta la decodifica, la CPU esegue l'istruzione. In generale, ciò comporta una modifica del contenuto dei registri. In particolare, cambierà almeno il contenuto del *program counter* (il registro della CPU che contiene l'indirizzo dell'istruzione successiva).

(a) *John Von Neumann (Budapest, 28 dicembre 1903 - Washington, 8 febbraio 1957).*



(b) *Struttura generale di un calcolatore di von Neumann.*

Figura 2.1: John Von Neumann

2.2 RAPPRESENTAZIONE DELLA MEMORIA DEL CALCOLATORE E TIPI DI DATO SEMPLICI

Può tornare utile immaginare la memoria di un computer come un foglio a quadretti (per gli attuali calcolatori, anche quelli presenti in ambienti domestici, il numero di "quadretti" è dell'ordine di 10^9). Ogni quadretto può contenere uno ed un solo "dato semplice" (ad esempio un carattere, un numero, o un qualsiasi altro *simbolo*).

Ogni dato (qualsiasi) occupa una certa quantità di quadretti (o celle). Ognuna di queste celle è identificata da un numero[1] progressivo (il suddetto *indirizzo*). Possiamo contrassegnare uno o più quadretti consecutivi con un nome simbolico (spesso risulta abbastanza scomodo usare gli indirizzi per individuare la posizione dei dati). In questo modo, identifichiamo dei "blocchi" di quadretti. Essi sono chiamati *variabili*. Le variabili vengono classificate in *tipi*.

Variabili

Il concetto di tipo, non è una caratteristica di tutti i *linguaggi di programmazione*. In C è necessario specificare il tipo di una variabile ma lo stesso non vale, ad esempio, per linguaggi come il Python. Ad ogni tipo di dato corrisponde una particolare quantità di memoria che il calcolatore *alloca* (alcuni tipi richiedono più celle di altri).

Tipi primitivi

I tipi *primitivi* in C sono:

- `char`: indica un carattere (lettera , cifra, segno...) ed occupa 1 cella;
- `int`: indica un numero intero (anche negativo, cioè appartenente a $\mathbb{Z}$);
- `float`: indica un numero decimale. Questa dovrebbe essere la rappresentazione di un numero reale all'interno del calcolatore. Tuttavia, per evidenti motivi, non ci è possibile memorizzare un numero con infinite cifre dopo la virgola. Per cui, il tipo `float` fornisce una rappresentazione approssimata di un numero irrazionale (o razionale periodico).

1 Il calcolatore non è in grado di "comprendere" la numerazione decimale. Esso è in grado di codificare solo istruzioni che gli vengano passate in linguaggio binario.

2.3 PROGRAMMAZIONE IMPERATIVA ED ASSEGNAMENTI

Assegnamento

Esistono diversi *paradigmi* (schemi generali) di programmazione; qui ci occuperemo della *programmazione imperativa*. In questo tipo di programmazione l'operazione principale è l'*assegnamento*, definito come il processo di inserimento dei dati nelle celle. Un esempio d'istruzione di assegnamento è la scrittura `x = 5303;` (in C). Un'istruzione d'assegnamento è formata da tre parti:

- Nome della variabile (ad esempio, `x`);
- Operatore di assegnamento (`=`);
- Valore della variabile (`5303`). Il valore della variabile deve essere compatibile con il suo tipo, altrimenti il computer segnalerà che c'è un errore.[2]

Sono ammessi assegnamenti anche del tipo `v = x;`, sempre a patto che le variabili `v` e `x` abbiano dei tipi compatibili tra di loro. Con questo assegnamento, anche la variabile `v` ora ha il valore `5303`.

Le variabili possono anche essere delle funzioni (matematiche) come, ad esempio, `y = v/10;`. Se `v` è un intero (poniamo `58`), il risultato sarà un intero (`5`). Se, invece, `v` è di tipo `float`, allora otterremo come risultato `5.8`. Possiamo anche scrivere degli assegnamenti in cui la stessa variabile compare da entrambe le parti. Ad esempio, `x = x+1;` è un assegnamento valido. Questo non fa altro che aggiornare il valore della variabile `x`.

Supponiamo ora di avere una variabile `beta` che sia stata dichiarata, ad esempio, di tipo `float`. Ammettiamo che essa contenga un certo valore e che noi vogliamo triplicare tale valore. Possiamo, in questo caso, scrivere `beta = 3*beta;`. Ciò è lecito: anche se `3` non è di tipo `float`, `3*beta` lo sarà.

Dichiarazione

Per comunicare al computer il tipo di variabile che intendiamo usare, dobbiamo effettuare la già citata operazione di *dichiarazione*. In C, essa assume la forma `float gamma;`, oppure:

```
float beta, gamma;
gamma = 1.0;
beta = gamma + 5;
```

Principali operatori algebrici

In generale, tutte le dichiarazioni vanno poste all'inizio del programma. Il C dispone anche di operatori aritmetici. I principali operatori sono: `+` (operatore somma), `-` (operatore differenza), `*` (operatore prodotto), `/` (operatore quoziente), `%` (operatore modulo, restituisce il resto di una divisione). Ad esempio:

```
int z;
z = 5%3;
```

restituisce un valore `z = 2`.

2.4 STRUTTURE DI CONTROLLO

L'ordine con cui vengono eseguite le istruzioni dipende dalle *strutture di controllo*. Esse sono, essenzialmente, gli assegnamenti ed altre poche istruzioni ed hanno il compito di "gestire" e, appunto, controllare il flusso delle informazioni durante l'esecuzione di un programma. Qualunque algoritmo può essere espresso per mezzo di tre strutture di controllo:

2 Le variabili di tipo `float` "contengono" un numero decimale. Si tenga presente che un assegnamento per una variabile di questo tipo dev'essere, ad esempio, `f = 23.934;`. *La virgola è di tipo anglosassone*. Il computer non riconoscerà un assegnamento del tipo `f = 23,934;`.

Codice 2.7: Struttura generale di un programma in linguaggio C.

```c
#include <stdio.h>

int main (void) {
    int x, y;
    float z, f;

    /*istruzioni */

    exit(...);
}
```

- Sequenza;
- Scelta;
- Iterazione.

2.4.1 Sequenza

La *sequenza* è determinata dall'ordine in cui vengono scritte le istruzioni. Per convenzione, la "lettura" avviene dall'alto in basso e da sinistra verso destra. Così, istruzioni scritte "in alto" verranno eseguite prima di istruzioni scritte "in basso".

2.4.2 Scelta

In C, la scelta si riconosce dalla preposizione `if`. Ad esempio:

```c
if ( x < 0 )
    x = -x;
```

Sintatticamente, le parentesi tonde sono obbligatorie. Esse racchiudono la condizione che deve essere verificata affinché il *corpo* della scelta venga eseguito. La condizione viene talvolta chiamata *test*. Se la condizione non è verificata, l'istruzione `x = -x;` (che nel nostro caso rappresenta il corpo della scelta) non viene eseguita. Si noti che è possibile trovare corpi che comprendano più di una istruzione. In questo caso, tali istruzioni vanno incluse tra parentesi graffe. Se ne vedranno degli esempi in seguito.

Operatori relazionali

Per il test, si dispone dei seguenti operatori relazionali: `<` (operatore minore), `<=` (operatore minore o uguale), `>` (operatore maggiore), `>=` (operatore maggiore o uguale), `==` (operatore uguale), `!=` (operatore diverso[3]).

È possibile trovare anche scelte che presentano le sintassi elencate nel riquadro 1 a fronte. Nel codice 2.3 nella pagina successiva, il corpo della scelta `if` è composto da tutte le istruzioni contenute tra parentesi graffe. Il calcolatore le esegue come se fossero una sola istruzione (naturalmente, rispettando la convenzione di eseguire prima quelle scritte" in alto").

Struttura caratteristica di un programma in C

In generale, un programma scritto in C ha una struttura generale caratteristica. Essa si ripresenta quasi uguale nella maggior parte dei programmi ed è mostrata nel codice 2.7.

3 In generale, il simbolo `!` nei test equivale ad una negazione. Quindi `!=` starebbe a significare "non uguale", cioè diverso.

Riquadro 1 Sintassi della scelta `if (...)else`

Codice 2.1

```
if ( x < 0 )
    x = -x;
else
    x = x+3;
```

Codice 2.2

```
if ( x < 0 )
    x = -x;
else
    x = x+3;
```

Codice 2.3

```
if ( x < 0 ) {
    x = -x;
    y = 3+z;
    e = y+1;
}
else
    ...
```

Codice 2.4

```
if (test1)
    if (test2)
        ...
    else
        ...
else
    ...
```

Codice 2.5

```
if (test1)
...
else
    if (test2)
        ...
    else
        ...
```

Codice 2.6

```
if (test1)
...
else if (test2)
        ...
    else
        ...
```

3 LEZIONE III

07/04/2011

3.0.3 Iterazione

Iterazione vuol dire *ripetizione*. Possiamo individuare due tipi di iterazioni. Più esattamente, la condizione verificata la quale un'iterazione continua può essere di due tipi:

- L'iterazione viene ripetuta un numero n di volte fissato a priori;
- Il numero di volte n per cui si ripete l'operazione viene determinato da come procede l'esecuzione (o l'iterazione stessa).

Nel linguaggio C, non esiste una forma che equivalga alla locuzione "ripeti l'operazione per n volte", ma bisogna ricorrere ad un'iterazione regolata da una determinata condizione.

Un espediente piuttosto frequente è mostrato nel codice 3.1. Si dichiara una variabile di tipo `int`. Si fa in modo che, ad ogni iterazione, tale variabile venga incrementata di un certo numero (ad esempio, di 1). Tale variabile fungerà, effettivamente, da *contatore*. Si fissa la condizione per la continuazione dell'iterazione (o *ciclo*) in modo tale che, quando il contatore raggiunge un dato valore, il ciclo si blocchi.

Contatore

Codice 3.1: *Tabella di conversione € - £.*

```
...
/*
** dichiaro la variabile (contatore) e la ini-
** zializzo, assegnandole un  valore iniziale
*/
int i = 0;
while ( i < 10 ) {
    printf("%d ... %f", i, i*1936.27);
    /* incremento il valore del contatore */
    i = i + 1;
}
...
```

L'iterazione, nel codice esempio 3.1, procederà finché la variabile `i` non avrà assunto il valore `10` (quindi, il ciclo si ripeterà 10 volte[1]). In questo esempio, il ciclo è stato introdotto introdotto usando la locuzione:

Il ciclo `while`

```
while (test) {
    ...
}
```

Essa inizializza un ciclo che procede finché il test espresso tra parentesi rimarrà verificato. Quando la condizione `i < 10` diventa falsa, l'esecutore salta il blocco `while` e procede con il comando che segue.

Le condizioni possono anche essere concatenate, cioè è possibile specificare più test all'interno di una sola coppia di parentesi tonde. Il C dispone degli *operatori logici* elencati nella tabella 3.1 a pagina 13.

Nel codice 3.1 è stato introdotto l'uso dei *commenti*. Un programma scritto in C può presentare dei commenti, parti di codice che il compilatore non

Commenti

1 Un errore molto frequente è impostare un ciclo di questo tipo dimenticandosi d'incrementare il contatore. Questo produrrebbe un ciclo infinito. Inoltre, il calcolatore non ha alcun modo di accorgersi di siffatti errori (si veda il paragrafo 10.1.3 a pagina 40).

cerca d'interpretare. Essi sono compresi tra *slash* ed *asterischi* (`/*commento */`) ed hanno lo scopo di migliorare la leggibilità del sorgente. Si può commentare il codice anche usando la forma `//commento`. La differenza tra le due forme è che, mentre nella prima il commento finisce dopo `*/`, nella seconda tutti i caratteri su una stessa riga (da `//` in poi) vengono ignorati al momento dell'esecuzione.

Il frammento di codice 3.1 nella pagina precedente, se opportunamente completato, stampa a schermo due colonne. A sinistra un numero intero da `0` a `9`, a destra lo stesso numero moltiplicato per `1936.27` (una tabella di conversione tra € e £). La funzione che stampa a schermo è `printf();`. Tale funzione può essere usata in diverse forme, riassunte nella tabella 3.2 nella pagina successiva.

printf();

Per leggere un valore inserito nello *standard input* (`stdin`), che nella maggior parte dei casi è la tastiera, si può usare la funzione `scanf();`. Tale funzione ammette le sintassi elencate nella tabella 3.3 a fronte. In tutti i casi il carattere `&` precede l'identificatore della variabile.

scanf();

3.1 STRUTTURE DATI

Una *struttura dati*[2] è un'entità usata per organizzare un insieme di dati all'interno della memoria del computer (ed eventualmente per memorizzarli in una memoria di massa). Detto in parole povere, è una rappresentazione schematica dei dati su cui s'intende lavorare. La scelta delle strutture dati da utilizzare è strettamente legata a quella degli algoritmi.

3.1.1 Vettori

Il *vettore* (o array) è un "oggetto" formato da una sequenza di valori omogenei (ossia, dello stesso tipo), una variabile in cui tutti i valori occupano posizioni adiacenti. Se si deve risolvere un problema che viene naturalmente rappresentato sotto forma vettoriale (o matriciale) questa struttura dati risulta estremamente utile. Per dichiarare un vettore, si usa la seguente sintassi: `double p[3];`.

Si supponga di aver bisogno di rappresentare la posizione di un punto (rispetto all'origine di un sistema d'assi cartesiani orientato) nello spazio tridimensionale. Risulta comodo usare un vettore che contenga dei valori Reali[3], come quello dichiarato poc'anzi, di nome `p`, che comprende tre variabili.

Di fatto, tale dichiarazione corrisponde a quella di tre variabili: `p[0]`, `p[1]`, `p[2]` (tutte di tipo `double`). Esse occupano posizioni contigue in memoria e condividono una parte del nome (oltre ad essere indicizzate). Volendo, possono anche essere usate separatamente come delle normali variabili. L'averle dichiarate sotto forma di un vettore permette di manipolarle meglio per operazioni vettoriali.

Nel seguente codice esempio, dopo aver dichiarato un vettore di lunghezza `100`, s'assegna alla sua e-esima coordinata il valore $e^2 - 3e$.

```
int v[100], e;
e = 0;
while ( e <= 99 ) {
    v[e] = e*e-3*e;
```

2 L'espressione "struttura dati" è grammaticalmente scorretta in italiano. La forma corretta sarebbe "struttura dei dati". Tuttavia, essa è una forma ereditata dall'inglese *data structures*, quindi viene accettata anche nella prima forma.

3 La migliore approssimazione che ottenibile di un numero $n \in \mathbb{R}$, è una variabile di tipo `double`.

Tabella 3.1: Operatori logici nel linguaggio C

Condizione	Operatore	Esempio
C_1 e C_2	`&&`	`( x >= 0 && x <= 10 )`
C_1 o C_2	`||`	`( x <= 0 || x >= 10 )`
Non C_1	`!`	`( !( x > y ))` cioè `( x < y )`

Tabella 3.2: Usi della funzione `printf();`.

Sintassi	Funzione
`printf("testo \n");`	Stampa un messaggio letteralmente. La combinazione `\n` equivale al carattere "a capo".
`printf("%d", num);`	Stampa il valore della variabile `num`, che dev'essere di tipo `int`.
`printf("%f", x);`	Stampa il valore della variabile `x`, che può essere di tipo `double` (in questo caso, la sintassi prevederebbe `%lf` e non `%f` tra virgolette) o `float`.
`printf("testo: %8.2f", x);`	Ogni sotto espressione introdotta da `%` indica il punto in cui sarà inserito il valore di una variabile. La sequenza compresa tra virgolette è seguita dall'elenco delle variabili cui si fa riferimento, separate da virgole. La combinazione `%8.2f` specifica che il valore della variabile `x` va stampato in modo che occupi almeno otto posizioni, delle quali due devono seguire il punto decimale.

Tabella 3.3: Usi della funzione `scanf();`.

Sintassi	Funzione
`scanf("%d", &n);`	Legge un valore di tipo `int` e lo assegna alla variabile `n`, che naturalmente dev'essere stata dichiarata di tipo intero.
`scanf("%f", &x);`	Legge un valore decimale e lo assegna alla variabile `x`, che dev'essere stata dichiarata di tipo `float`.
`scanf("%lf", &y);`	Legge un valore di tipo decimale e lo assegna alla variabile `y`, che dev'essere stata dichiarata di tipo `double`.

```
    e = e + 1;
}
```

3.2 FUNZIONI

In matematica, una *funzione* (o *applicazione*) f è una *relaizone* definita da:

- Un insieme X detto *Dominio* della funzione f;
- Un insieme Y detto *Codominio* della funzione f;
- Una legge tale per cui $\forall x \in X, \exists!\ y \in Y \mid y = f(x)$.

Si consideri una funzione tale che:

$$y = \begin{cases} abs : \mathbb{R} \mapsto \mathbb{R} \\ abs(x) = |x| \end{cases}$$

All'interno di un programma, si possono definire delle parti di codice che eseguono delle operazioni specifiche (*funzioni*, appunto). Per fare ciò, bisogna specificare un Dominio, un Codominio e la forma della funzione (la sequenza di istruzioni che la compongono). Successivamente, la funzione sarà disponibile per essere richiamata in altre parti del programma.

Il codice 3.2 mostra la precedente funzione stesa in linguaggio C. Si noti che, dopo averla dichiarata, la si può richiamare come a riga 16. Si tenga inoltre presente che ogni programma in C *deve* contenere una funzione `main()`.

Codice 3.2: *Definizione e chiamata della funzine* `abs()`.

```
/* codominio_nome della funzione_(dominio) */
double abs ( double x ) {
    /*
    ** Il corpo di una funzione va sempre posto
    **fra parentesi graffe.
    */
    if ( x < 0 )
        return -x;
    else
        return x;
}

int main ( int argc, char *argv[] ) {
    double v, y;
    scanf("%lf", &v);
    y = abs(v);
    printf("%lf", y);
    exit(EXIT_SUCCESS);
}
```

Si considerino delle funzioni con due (o più) parametri formali, come la seguente:

$$\begin{cases} max : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R} \\ max(x, y) = \begin{cases} x, & \text{se } x > y \\ y, & \text{se } x < y \end{cases} \end{cases}$$

Essa trova il massimo tra due numeri. Il codice 3.3 a fronte, ne mostra la stesura in C.

Codice 3.3: *Funzione `max()`, con due argomenti.*

```
double max ( double x, double y ) {
    if ( x < y )
        return y;
    else
        return x;
}
```

Tabella 3.4: Opzioni delle funzioni `printf();` e `scanf();`

Opzione	Funzione
`%d`	Stampa/legge variabili di tipo `int`.
`%f`	Stampa/legge variabili di tipo `float`.
`%lf`	Stampa/legge variabili di tipo `double`.
`%c`	Stampa/legge variabili di tipo `char`.
`%s`	(Con `scanf()`) Legge stringhe di caratteri e le salva in un array.

4 LEZIONE IV

14/04/2011

4.1 ESERCIZIO

Si vuole calcolare il valore di π. È noto che:

$$\frac{\pi}{4} = \sum_{k=0}^{+\infty} (-1)^k \frac{1}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} \cdots$$

Questo metodo non può essere steso come *algoritmo* (che è **finito** per definizione[1]). Tuttavia, ci si può accontentare di approssimazioni di π che discostino dal valore reale per una quantità finita piccola a piacere facendo eseguire al calcolatore un numero finito (se pur molto grande) di iterazioni. La sommatoria, allora, diverrà:

$$\frac{\pi}{4} \approx \sum_{k=0}^{n \in \mathbb{N}} (-1)^k \frac{1}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} \cdots + \frac{(-1)^n}{2n+1}.$$

Un algoritmo come quello del codice 4.1 calcolerà un valore approssimato del valore di π. Chiaramente, la precisione del valore calcolato dipenderà dalla grandezza di n. Infatti

$$\lim_{n \to \infty} \sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = \frac{\pi}{4}.$$

Codice 4.1: *Calcolo del valore approssimato di π.*

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main ( int argc, char *argv[] ) {
    int n, i = 0;
    double somma = 0.0;
    printf("Inserisci n: ");
    scanf("%d", &n);
    while ( i <= n ) {
        if ( i%2 == 0 )
            somma = somma + (1/(2*i + 1));
        else
            somma = somma - (1/(2*i + 1));
        i = i + 1;
    }
    printf("Pi Greco = %lf", somma*4);
    exit(EXIT_SUCCESS);
}
```

1 Un algoritmo è un procedimento che consente di ottenere un risultato eseguendo, in un determinato ordine, dei passi semplici (scelti da un insieme finito). Pertanto:

- La sequenza di istruzioni deve essere finita (*finitezza*);
- Essa deve portare ad un risultato (*effettività*);
- Le istruzioni devono essere eseguibili materialmente (*realizzabilità*);
- Le istruzioni devono essere espresse in modo non ambiguo (*non ambiguità*).

I passi costituenti di un algoritmo devono essere "semplici", nel senso di "non ambigui".

5 LEZIONE V

21/04/2011

5.1 ANCORA SULLE FUNZIONI

La funzione è una specie di "scatola" cui si associa un nome simbolico: vi entrano dei dati e ne esce il risultato della loro elaborazione. Quest'ultima avviene per mezzo delle istruzioni che ne compongono il corpo.

L'istruzione `return`

In una funzione, l'istruzione `return` è sempre l'ultima ad essere eseguita e ne indica la fine. Si noti che ciò non vuol dire essa che debba essere l'ultima ad essere scritta. Se è inclusa in una scelta, ad esempio, non è affatto detto che debba essere eseguita. È lecito, dunque, che dopo `return ...;` vi siano altre righe di codice.

Quando una funzione viene eseguita, il processo principale si "blocca" in attesa del suo risultato. Tutte le variabili create durante questo lasso di tempo vengono cancellate quando essa termina (all'istruzione `return`, appunto).

Argomenti formali e reali

Nella definizione della funzione, le variabili che si trovano all'interno delle parentesi tonde sono dette *argomenti formali*. Nel momento in cui la si richiama in qualche altra riga del codice, le si passano degli *argomenti reali*. Il codice 5.1 è un esempio di chiamata della funzione `flip()` (all'interno della funzione `main()`). Qui, le variabili `a`, `b` e `c` sono gli argomenti reali della chiamata della funzione `flip()`. Si noti che essi sono stati dichiarati in modo da essere compatibili con i tipi richiesti dagli argomenti formali.

Codice 5.1: *Chiamata della funzione `flip()`.*

```
double flip (int x, double y, int z ) {
    /* istruzioni */
    return ... ;
}

int main ( int argc, char *argv[] ) {
    int a, c;
    double b, x;
    x = flip( a, b, c );
    ...
}
```

Variabili locali e globali

Le variabili definite in qualsiasi funzione si prendono il nome di *variabili locali* della funzione `x()` (dove `x()` è il nome della stessa: `flip()`, ad esempio). In C, esse si distinguono dalle *variabili globali*. Queste ultime, infatti, sono quelle dichiarate al di fuori di tutte le funzioni e restano a disposizione di ognuna di esse per tutta la durata del programma (cioè della funzione `main()`). Ogni funzione può modificare il valore di una variabile globale, mentre non è possibile richiamare direttamente[1] una variabile della funzione `flip()` dalla `main()`, ad esempio.

Passaggio per valore

La comunicazione tra funzioni avviene tramite un procedimento chiamato *passaggio per valore* (descritto nel prossimo esempio). Una funzione riceve il valore di una variabile come parametro reale, ma *non* può modificare

1 Lo si può però fare tramite i puntatori (vedi il paragrafo 6.3 a pagina 25).

Riquadro 2 I caratteri in C.

Codice 5.2 *Il tipo* `char`.

```
char x;
char p[10];
p[3] = 'n';
```

Codice 5.3

```
char x = 'a';
printf("%c", x + 1);
```

il valore assegnato alla stessa[2]. Quando una funzione viene eseguita, il calcolatore le riserva uno spazio chiamato *record di attivazione*.

Si supponga di voler scrivere un programma che calcola la somma dei primi n elementi di un array:

```
#include <...>
...
double somma ( double v[], int n ) {
    int i = 0;
    double s = 0.0;
    while ( i < n) {
        s = s + v[i];
        i = i + 1;
    }
    return s;
}
...
int main ( int argc, char *argv[] ) {
    double m[10], p[20], v, w;
    ...
    v = somma( m, 10 );
    w = somma( p, 15 );
    ...
}
```

Si noti che tale programma funziona se e solo se, quando si chiama la funzione `somma()`, il secondo argomento è minore o uguale alla lunghezza del vettore passato come primo argomento. In caso contrario, si otterrà soltanto un errore di *segmentazione*.

Il codice 5.2 nel riquadro 2, introduce ora un nuovo tipo di variabile, che corrisponde al "simbolo".

I vettori di caratteri costituiscono la rappresentazione delle parole in C. Le variabili di tipo `char` si dichiarano racchiudendo il valore da assegnare tra singoli apici. Anche i numeri possono essere considerati dei caratteri, purché racchiusi tra apici. Dopo aver dichiarato un numero come carattere, tuttavia, non è possibile eseguire le consuete operazioni algebriche su di esso.

Rappresentazione dei caratteri

Nel calcolatore, ad ogni carattere è assegnata una sequenza di cifre binarie (in genere, 8 cifre) che è effettivamente un numero. Pertanto, l'espressione `p[3]*4` ha significato, ma non è quello che ci si aspetterebbe. La codifica numerica dei caratteri, tuttavia, può consentire delle operazioni interessanti. Le lettere dell'alfabeto, ad esempio, sono numerate in ordine progressivo[3]. Con un programma simile a quello del codice 5.3, il calcolatore stamperà a schermo il carattere `b`. Tale rappresentazione dei caratteri permette, inoltre, di ordinare alfabeticamente le parole.

2 Questo non è sempre vero. A causa della loro stretta relazione coi puntatori, è possibile che il valore di una variabile dichiarata come array passata come argomento reale venga modificata durante l'esecuzione di una funzione (si veda il paragrafo 6.3 a pagina 25).

3 Le lettere maiuscole sono poste in ordine progressivo tra di loro, così come quelle minuscole. Tuttavia, le rappresentazioni di `a` e `B` non hanno differiscono tra di loro per un unità.

5.1.1 Filtri

La funzione `getchar();`

L'istruzione `x = getchar();` comunica all'esecutore di leggere il carattere successivo dallo standard input (in genere, la tastiera) e di toglierlo dalle serie dei "caratteri in entrata". `getchar()` è una funzione che non ammette parametri e ha come unico risultato l'assegnamento del carattere letto alla variabile cui è associata (nella fattispecie, `x`).

La funzione `putchar(x);`

La funzione `putchar(x);`, invece, stampa sullo *standard output* (`stdout`, in genere il monitor) il valore della variabile `x`.

Il programma nel codice 5.4 "fa l'eco" di quanto riceve dallo standard input. Esso appartiene alla famiglia dei *filtri*, cioè programmi che leggono dei caratteri ed eseguono delle operazioni su di essi. `EOF` è una sequenza di caratteri speciale (del linguaggio C) che sta per `End Of File`. Per i sistemi UNIX, la sequenza riconosciuta come `EOF` è la combinazione di tasti `CTRL+D`.

Codice 5.4: *Esempio di filtro.*

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main ( int argc, char *argv[] ) {
    int c;
    c = getchar();
    while ( c != EOF ) {
        putchar(c);
        c = getchar();
    }
exit(EXIT_SUCCESS);
}
```

6 LEZIONE VI

28/04/2011

6.1 STRUTTURE DATI

6.1.1 Grafo

Un *grafo* (figura 6.1) è un oggetto $G = (V, E)$ composto da *vertici* $v_n \in V$ e *lati* $e_m \in E$. Si ha che:

$$V = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$
$$E \subseteq V \times V$$

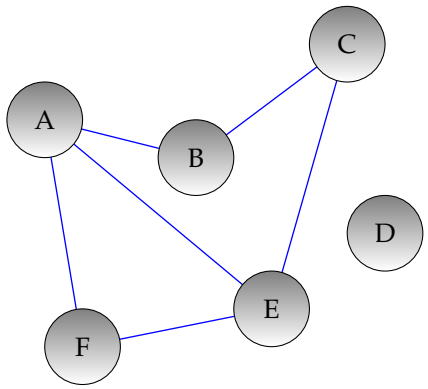


Figura 6.1: Esempio di grafo.

L'insieme dei lati è un sottoinsieme del prodotto cartesiano dei punti. Ognuno di essi è *univocamente* determinato da una coppia di punti (se non è orientato). Un lato, quindi, è un segmento non orientato completamente individuato da una coppia di punti. Se consideriamo dei segmenti orientati, probabilmente, il grafo sarà una funzione di grandezze diverse. Bisogna cioè aggiungere altre informazioni.

Supponiamo di avere segmenti non orientati. Siano $v_1, v_2 \in V$ per rappresentare un segmento da v_1 a v_2, allora $\overline{v_1 v_2}$ e $\overline{v_2 v_1}$ coincidono. Nel caso di segmenti orientati, tuttavia, possiamo stabilire, per convenzione, che il primo punto sia l'origine del segmento, il verso sia indicato dal secondo punto e che la differenza tra i vertici permetta di stabilire il modulo del vettore. Quindi, per un vettore orientato da v_1 a v_2 ci basterà scrivere $\overline{v_1 v_2}$ (oppure $\overrightarrow{v_1 v_2}$).

6.1.2 Strutture

Si supponga di voler creare un programma che tenga traccia della carriera scolastica di alcuni studenti. Innanzitutto, bisogna cercare di selezionare i dati essenziali ai fini del programma. Si può immaginare lo studente come un ente univocamente determinato da tre variabili. Si dichiarano, allora[1]:

```
char nome[24];
int n_matricola;
int voti[6];
```

Queste variabili, tuttavia, sono ancora indipendenti tra di loro. Dal momento che si riferiscono ad una stessa persona, sarebbe più utile poterle trattare come se fossero tutte "legate". È questo il concetto di *record*. Un record permette di trattare più variabili non omogenee, come se fossero aggregate tra di loro.

Record

1 Si può anche dichiarare `char n_matricola[6];` ma è molto più semplice che a rappresentare la matricola sia un numero (intero).

Codice 6.1: *Definizione di una struttura e dichiarazione di variabili.*

```
1 struct studente {
      char nome[24];
3     int matricola, voti[6];
  } maria, antonio;
5
  int main ( int argc, char *argv[] ) {
7     struct studente giuseppe, giovanni;
      ...
9 }
```

Nella fattispecie, il record `studenti` contiene tre *campi*. Nel linguaggio C, il record prende il nome di *struttura*. La definizione delle strutture segue generalmente le prime direttive in cima al codice sorgente (`#include <...>`) in modo che risulti ben visibile. Si noti che dichiarare una struttura equivale, di fatto, a definire un nuovo tipo di variabile.

Nel codice 6.1 `studente` è l'*identificatore* (o *tag*) della struttura[2]. All'interno delle parentesi graffe (obbligatorie) si dichiarano le variabili che formeranno i campi della struttura. È possibile dichiarare delle variabili di tipo `struct studente` immediatamente dopo la definizione della struttura (come `maria` e `antonio` nel codice 6.1). Alternativamente, è consentito farlo in qualsiasi altra riga di codice, usando la sintassi descritta nel codice 6.1 per la dichiarazione di `guseppe` e `giovanni`.

`typedef`

Il linguaggio C consente di assegnare un "sinomimo" a `struct studente` tramite la funzione `typedef tipo_1 tipo_2;`:

Codice 6.2: *`typedef` e assegnamenti.*

```
  typedef struct studente Studente;
2
  int main ( int argc, char *argv[] ) {
4     Studente maria, antonio;
      maria.matricola = 73659;
6     maria.voti[2] = 27;
      ...
8 }
```

In questo modo, si possono dichiarare delle variabili di tipo `struct studente`, tramite la parola `Studente`. Volendo richiamare (o assegnare un valore ad) un campo della struttura, la sintassi è riportata nelle righe 6 e 7 del codice 6.2. La regola è: *nome.campo* `=` *valore*`;`.

Si tenga presente che, avendo dichiarato due record, `a` e `b`, è lecito operare un assegnamento nel modo usuale `a = b;`. Tuttavia, non si possono confrontare due strutture con la scrittura `a == b;`: bisogna confrontare le strutture campo per campo.

Direttiva `#define`

Spesso può risultare conveniente specificare una direttiva[3] `#define` all'inizio del codice sorgente come mostrato nel codice 6.3 nella pagina successiva. Dopo aver specificato questa direttiva, il compilatore sostituirà il valore `30` ogni qualvolta incontrerà il carattere `L`[4]. Tale direttiva renderà il programma molto più facile da modificare.

Si supponga di voler creare una struttura che rappresenti un punto nello spazio bidimensionale. Oltre alla rappresentazione vettoriale, è possibile utilizzare un record composto da due campi, come nel codice 6.4 a fronte. Inoltre, poiché un rettangolo (parallelogramma retto) è univocamente individuato dagli estremi della sua diagonale, ci bastano due punti.

2 Non è strettamente necessario introdurre la tag ma, per motivi di leggibilità, è fortemente consigliato.
3 Le direttive *non* sono istruzioni, quindi non bisogna concludere la riga con `;`.
4 Il carattere deve essere una *unità logica*, cioè, il compilatore non sostituirà il valore `30` a `L` se esso è (ad esempio) all'interno di una parola.

Codice 6.3 *La direttiva* `#define`.

```
#include <...>
#define L 30

struct nome {
    char name[L];
};
```

Codice 6.4 *Punto e rettangolo.*

```
struct punto {
    double x;
    double y;
}
struct rettangolo {
    struct punto a_sx;
    struct punto b_dx;
}
```

6.2 STRINGHE

In C non esiste un tipo di variabile specifico per gestire le *stringhe*[5]. Le stringe vengono trattate come sequenze di caratteri. Per memorizzarle, quindi, si renderà necessario dichiarare un array di caratteri. In C, ogni stringa termina con un carattere speciale detto *terminatore* (rappresentato con `\0`). Si noti che anch'esso occupa lo spazio di un carattere in memoria.

La funzione `strcpy();`

La funzione `strcpy(dest[], sorg[]);` copia una stringa da un array di caratteri, specificato come secondo argomento reale (`sorg[]`), ad un altro, indicato dal primo parametro reale (`dest[]`). Si può sfruttare tale funzione anche per effettuare un assegnamento, come nel codice 6.5. In alternativa, è possibile usare la funzione `scanf();` con l'opzione `%s` (vedi la tabella 3.4 a pagina 16) per assegnare ad un array di tipo `char` stringhe immesse nello dallo `stdin`.

Codice 6.5: *`strcpy();` e `scanf();`.*

```
struct studente {
    char nome[24];
    int matricola, voti[6];
} maria, antonio;

int main ( int argc, char *argv[] ) {
    strcpy(maria.nome, "Maria");
    scanf("%s", &antonio.nome);
    ...
}
```

6.3 PUNTATORI

Come già accennato nel paragrafo 2.2 a pagina 6, ogni variabile occupa una determinata porzione di spazio e una posizione identificata da un indirizzo (che è praticamente un numero). Ora, è possibile manipolare delle variabili in due modi differenti ma equivalenti:

DIRETTAMENTE: tramite assegnamenti che operino su di esse;

INDIRETTAMENTE: conoscendo i loro indirizzi[6].

Nel codice d'esempio seguente, si è dapprima assegnato alla variabile `p` l'indirizzo di `x`. Successivamente si dà ad `y` il valore di `x` usando il suo indirizzo. Nella terza istruzione `x` assume il valore `0`, sempre per mezzo del suo indirizzo memorizzato nella variabile `p`.

5 In Informatica, una stringa è una sequenza di caratteri (ad esempio, una parola).

6 L'operatore & restituisce l'indirizzo di una variabile. L'operatore *, invece, ci permette di modificare il valore di una variabile conoscendo il suo indirizzo.

Riquadro 3 Passaggio per valore

Codice 6.6 *x non viene modificato.*

```
int function (int a, int b) {
    a = 2*b;
    ...
}
int x, y;
c = function(x, y);
```

Codice 6.7 *x viene modificato.*

```
int function (int *a, int b) {
    *a = 2*b;
    ...
}
int x, y;
c = function(&x, y);
```

```
p = &x;
y = *p;
*p = 0;
```

Si badi che nella dichiarazione di un vettore, sia esso `int p[10];`,la lettera `p` è, in sostanza, un puntatore alla prima variabile del vettore. In altre parole `p` equivale a `&p[0]`.

Com'è stato già detto nel paragrafo 5.1 a pagina 19, una funzione non può modificare il valore del suo parametro reale tramite il passaggio per valore. Si consideri il riquadro 3. Nel codice 6.6, la funzione non modificherà il valore di `x`. Nel 6.7, invece, il valore di `x` verrà modificato anche quando la funzione sarà terminata perché la funzione "ha accesso" al valore di `x` tramite il suo indirizzo.

Per quanto detto sui vettori, passando come parametro il nome di un vettore in una funzione simile il valore della variabile verrà modificato.

7 LEZIONE VII

05/05/2011

7.1 NOTAZIONI ALGEBRICHE

Si consideri una qualsiasi operazione algebrica. Fino ad ora, si sono adoperate soltanto sintassi in cui l'operatore s'inserisce tra i due operandi (ad esempio, `3+4`). Questo tipo di notazione si dice *infissa*. In questa forma, espressioni complicate o lunghe richiedono l'uso di parentesi (come `5*(3+4)`).

Si prenda la seguente espressione: `5*{[(9+8)*(4+6)]+7}` (notazione infissa); esiste un altro modo di rappresentarla. È, infatti, possibile usare la forma *postfissa* (o polacca) e riscriverla in questo modo: `5 9 8 + 4 6 + * 7 + *`. La notazione postfissa prevede, come si evince dall'esempio, che l'operatore algebrico segua i due operandi. Nel caso più semplice, `3+4` diventa `3 4 +`. L'espressione si risolve nel modo usuale:

```
5 9 8 + 4 6 + * 7 + * =
5 17 10 * 7 + * =
5 170 7 + * = 5 177 * = 885
```

7.2 STRUTTURE DATI

7.2.1 Stack

Gli aspetti caratteristici di una struttura dati dati sono:

- I valori che riesce ad assumere (quindi le grandezze che può rappresentare);
- Le operazioni che posso compiere sulla struttura.

Per i tipi primitivi, le operazioni definite sono piuttosto implicite e date per scontate fino a questo momento. In ogni caso, si tratta di operazioni algebriche di base (somma, prodotto, resto...). Sfruttando la notazione postfissa, è possibile creare una struttura dati che definisce un particolare procedimento per risolvere un espressione.

Operazioni `pop` e `push`

Si immagini di avere una *pila* (in inglese, *stack*) di dischetti. Ogni dischetto può memorizzare uno ed un solo numero. Sulla pila è possibile compiere solo due operazioni (vedi la figura 7.1 nella pagina seguente):

- `push`: mettere un dischetto in cima alla pila;
- `pop`: togliere un dischetto dalla cima della pila.

Questo procedimento definisce, per i dati, una disciplina del tipo LIFO (*Last In Firs Out*). L'ultimo elemento inserito, cioè, è il primo ad essere prelevato.

Ora, si vuole creare un algoritmo che permetta di risolvere delle espressioni scritte in notazione postfissa sfruttando le operazioni concesse dagli stack. Si potrebbe immaginare di memorizzare nei dischetti numeri immessi (`push`) finché non si legge un operatore matematico. A questo punto, si estraggono i due dischetti che si trovano in cima alla pila (`pop`) e si esegue su di essi l'operazione richiesta dall'operatore immesso. Successivamente, si memorizza il risultato dell'operazione su un dischetto e lo si mette in cima alla pila (`push`).

Tabella 7.1: La tabella mostra il contenuto dello stack ad ogni iterazione (tempo).

Tempo	0	1	2	3	4	5	6	7	8	9	10
Pila	5	9	8	17	4	6	10	170	7	177	855
		5	9	5	17	4	17	5	170	5	
			5		5	17	5		5		
						5					

Prototipi delle funzioni `pop()` e `push()`

In questo modo è possibile calcolare l'espressione precedente. In questo caso, la tabella 7.1 mostra i valori presenti nello stack durante l'esecuzione dell'algoritmo.

Si supponga che sia stato definito un tipo di dato `stack` in linguaggio C. Si crea, allora, un prototipo della funzione `push`: `stack push (stack s, int v )`. Si può assumere che la funzione `push` riceva come argomento un numero (nella fattispecie, un intero) e lo stack su cui inserirlo, e restituisca uno stack diverso. La funzione `pop`, produrrebbe un nuovo stack e un intero. Tuttavia, in C, una funzione può produrre un solo valore. Allora, si ricorre ad un piccolo "trucchetto": definendo un solo stack, tutte le funzioni possono solo lavorare su di esso e non c'è più bisogno d'inserirlo né tra gli argomenti della funzione, né nel suo codominio.

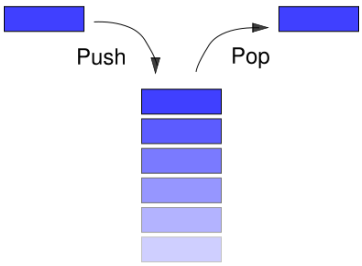


Figura 7.1: Rappresentazione delle funzioni `pop()` e `push()` su uno stack.

Ora, bisogna costruire un tipo di dato che permetta di rappresentare uno stack. Esistono diverse opzioni. Una delle più comode, per il momento, resta costruire un array (vedi anche il paragrafo A.1 a pagina 81). Esso, tuttavia, è una struttura dati non dinamica: la sua lunghezza non cambia. Si tratterà questo problema più avanti (nel paragrafo 7.3 a fronte). Per ora, si assuma di aver dichiarato un array di dimensioni sufficienti per contenere tutti i dati inseriti. Si può pensare di riempire il vettore dalla posizione più piccola (`0`) e continuare progressivamente verso quella più grande. Per tenete traccia del primo elemento vuoto del vettore, c'è bisogno di dichiarare una variabile ausiliaria (in questo caso di tipo `int`). Il codice 7.1 esemplifica quanto detto finora. È importare tenere presente, tuttavia, che la funzione `push` dà errore (di *segmentazione*) se `valori.indice == N`. Analogamente, la funzione `pop` dà errore se il vettore `valori.val` è vuoto. Infatti, se `valori.indice == 0`, la funzione `pop` dà errore (ancora di segmentazione). Vi si può rimediare semplicemente introducendo delle scelte.

Codice 7.1: *Costruzione di uno stack. In questo esempio s'è usato un record per raggruppare la pila e l'intero che punta al primo elemento vuoto.*

```
/* stack */
struct pila {
    char val[N];
    int indice;
} valori;

void push (char v) {
    valori.val[valori.indice] = v;
    valori.indice++;
}

/* senza argomenti */
char pop (void) {
```

```
14      valori.indice--;
        return valori.val[valori.indice];
16  }
```

7.3 ALLOCAZIONE DINAMICA DELLA MEMORIA

Si è già detto che il vettore, come tutte le strutture dati viste finora, è un modello di dati *statico*. Una volta dichiarato, ha una lunghezza fissa. In C, tuttavia, è possibile riservare più spazio ad una o più variabili durante l'esecuzione di un programma.

La funzione `malloc();`

La funzione `malloc(size_t N);` permette di riservare `N byte` per l'assegnamento di una variabile. Anche se non è stata ancora introdotta la locuzione `size_t N`, si vede chiaramente che si riferisce al numero di `byte`[1] che si chiede di riservare. Inoltre, `N` è sempre (strettamente) maggiore di zero. La funzione `malloc(size_t N);`, in realtà, restituisce un puntatore di tipo speciale `void` (`void * malloc(size_t N);`). Questo crea un'inconsistenza formale, poiché tale tipo non corrisponde a quello del puntatore che si è dichiarato. Anche se il calcolatore tollera tale inconsistenza, è sempre meglio operare un `cast`, come nel codice 7.2, specificando tra parentesi il tipo di puntatore davanti alla funzione `malloc(...);`: `n = (tipo *)malloc(sizeof(tipo));`. La funzione `sizeof(tipo);` (alle righe 9 e 14) restituisce il numero di `byte` occupato dal tipo di dato specificato come argomento.

`malloc();` e gli array

La funzione `malloc();` assume una forma particolarmente comoda per la dichiarazione e l'uso di array. Nel codice 7.2 (riga 14), ad esempio, si chiede riservare lo spazio per `100` interi. Esso sarà contiguo e quindi adatto ad essere indicizzato tramite un vettore. La stretta relazione tra il nome di un array e i puntatori cui si è accennato in precedenza permette di effettuare degli assegnamenti come quelli nell'ultima riga.

Codice 7.2: *Uso della funzione `malloc();`. Si faccia particolare attenzione a non dimenticare d'inserire la scelta immediatamente dopo aver richiamato tale funzione.*

```
    struct nodo {
2       int val;
        char c;
4   } x, *p, *n;
    int *pn;
6   ...
    p = &x; /* assegno a p l'indirizzo di x */
8   (*p).val = 0; /* assegno 0 a x.val */
    n = (struct nodo *)malloc(sizeof(struct nodo));
10  if ( n == NULL )
        exit(EXIT_FAILURE);
12  (*n).val = 10;

14  pn = (int *)malloc(100*sizeof(int));
    if ( pn == NULL )
16      exit(EXIT_FAILURE);
    pn[10] = 1;
```

Valore NULL

Bisogna prestare particolare attenzione al fatto che, qualora la funzione `malloc(...);` non trovi abbastanza spazio contiguo da allocare, ritorna con il valore `NULL`, valido come puntatore, ma non come indirizzo. Allora, si

1 Un `byte` può contenere un carattere. Generalmente, un intero occupa `4 byte`. Questo, tuttavia, non è sempre vero.

Figura 7.2: Lista concatenata monodirezionale. I campi in blu sono puntatori.

fa seguire ad ogni istruzione `malloc(...);` la scelta[2] mostrata nel codice 7.2 nella pagina precedente.

7.4 STRUTTURE DATI

7.4.1 Liste Concatenate

Con questi strumenti è possibile costruire qualcosa che somigli più di un array ad uno stack. L'idea è di comporre una *lista concatenata*, ossia una sequenza di strutture (record) con gli stessi campi che dal punto di vista logico sono ordinate tramite un dato criterio. Esse sono generalmente "collegate" da puntatori (vedi la figura 7.2). Le strutture che compongono una lista, infatti, non occupano posizioni contigue in memoria ma, nella maggior parte dei casi, sono "sparpagliate". Qui intervengono i puntatori: ogni struttura dovrà contenere un campo che punti all'elemento successivo della lista. L'ultimo elemento avrà il campo puntatore contenente il valore `NULL` (per segnare la fine della lista). Questi accorgimenti consentono d'inserire altri record in qualsiasi posizione della lista.

Si può ora rappresentare uno stack tramite una lista concatenata. Ovviamente, servirà un puntatore "esterno" (chiamato generalmente `*testa`) che punti al nodo che si trova "in alto" nella pila, analogamente a quanto si è fatto con gli array.

```
struct nodo {
    int v;
    char c;
    struct nodo *next;
}
```

2 Il professor Bernardinello ha tenuto a specificare che la mancata introduzione della scelta dopo tale funzione verrà considerata errore.

8 LEZIONE VIII

12/05/2010

8.1 COSTRUZIONE DI UNA LISTA CONCATENATA

Come già accennato, una lista concatenata è una sequenza di oggetti (chiamati anche *nodi*) nella quale è definita una relazione d'ordine. Il criterio d'ordinamento è dato dalla successione dei puntatori. In una lista concatenata, per raggiungere l'n-esimo elemento bisogna percorrere tutti gli $n-1$ elementi precedenti.

Per costruire concretamente una lista concatenata c'è bisogno, oltre che degli elementi della lista, di una variabile (puntatore) che contenga l'indirizzo del primo elemento. In generale, risulta comodo chiamare questa variabile `*testa`, come nel codice 8.1. All'inizio la lista è vuota ma `testa` deve avere un valore legittimo. Per convenzione si pone, in questo caso, `testa = NULL`. Si noti che l'assegnamento `(*pn).next = testa;` equivale a `pn->next = testa ;`: usare una delle due sintassi è indifferente ai fini della compilazione del programma.

Codice 8.1: *Costruzione di una lista concatenata.*

```
struct nodo {
    int v;
    char c;
    struct nodo *next;
} *testa, *pn;

typedef struct nodo Nodo;
testa = NULL;

pn = (Nodo *)malloc(sizeof(Nodo));
testa = pn;
int i = 1;
while ( i < 10 ) {
    pn = (Nodo *)malloc(sizeof(Nodo));
    if ( pn == NULL )
        exit(EXIT_FAILURE);
    (*pn).next = testa;
    testa = pn;
    i++;
}
```

A differenza dei vettori, i nodi della lista concatenata non sono indicizzati tramite dei numeri ma, come già accennato, tramite dei puntatori. Volendo, allora, scorrere tutti gli oggetti presenti fino all'ultimo, si scriverà un codice simile al seguente:

```
pn = testa;
while ( pn != NULL ) {
    /* eventuali istruzioni */
    pn = pn->next;
}
```

Si tenga presente che, se la lista è vuota, il ciclo fallisce perché la condizione `pn == NULL` è verificata fin dal primo elemento.

8.2 ESEMPI

I codici 8.2, 8.3 e 8.4 sono esempi di come inserire nodi in una lista concatenata rispettivamente all'inizio, alla fine ed in una posizione intermedia.[1]

Codice 8.2: *Inserimento in testa*

```
pn = (Nodo *)malloc(sizeof(Nodo));
if ( ... )
    exit(...);
pn->next = testa;
testa = pn;
```

Codice 8.3: *Inserimento in coda*

```
/* pi funge da contatore */
pi = testa;
/* scorro la lista finche' non finisce */
while( pi->next != NULL)
    pi = pi->next;
/*
** assegno all'ultimo puntatore l'indizizzo
** del nodo pn (creato con malloc();)
*/
pi->next = pn;
/*
** Ora pn e' l'ultimo, quindi il suo campo
** .next dev'essere NULL
*/
pn->next = NULL;
```

Codice 8.4: *Inserimento intermedio*

```
pi = testa;
while( pi->next /* condizione */)
    pi = pi->next;

pn->next=pi->next;
pi->next = pn;
```

1 Nell'esempio 8.4 non è stata inserita la `/*condizione */` perché dipende dal criterio che si vuole usare per inserire il nodo.

9 LEZIONE IX

19/05/2011

9.1 RAPPRESENTAZIONE DEI NUMERI NEL CALCOLATORE

Un sistema di rappresentazione numerica può essere *posizionale* o *non posizionale*. Nel primo caso, ogni cifra assume un peso diverso a seconda della posizione in cui è scritta. Una rappresentazione numerica è anche caratterizzata dalla *base*, che fissa il numero massimo di simboli differenti che consentiti per rappresentare le cifre; il sistema numerico occidentale è *decimale posizionale*. Pertanto:

- La base (`10`) stabilisce il numero di simboli differenti consentiti per rappresentare le cifre (`0, ... , 9`).
- Il fatto che sia posizionale ci garantisce che, ad esempio, $123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 \neq 321 = 3 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0$. Ciò accade proprio perché le cifre hanno pesi differenti a seconda della loro posizione, infatti:

  $$4814 = 4 \cdot 10^3 + 8 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0.$$

  In generale, essendo b la base della rappresentazione e c_i (con $i \in \{1, \dots, b\} \cap \mathbb{N}$) i rappresentatori delle cifre, dato $z \in \mathbb{N}$ si ha che, $\forall i, \dots, j \in \{1, \dots, b\} \cap \mathbb{N}$:

  $$c_{i_z} \dots c_{j_1} = c_{i_z} 10^{z-1} + \dots + c_{j_1} 10^0 = \sum_{e=1}^{z} c_{\{i,\dots,j\}_e} \cdot b^{e-1}.$$

9.1.1 Esempi

Si considerino i seguenti esempi. Le basi *ottale*, *esadecimale* e *binaria* non sono scelte a caso; esse, infatti, sono molto usate nel campo dell'informatica. Si noti che, nel caso di rappresentazioni non decimali, è buona norma specificare la base a pedice.

BASE 8

- Simboli: `0, 1, 2, 3, 4, 5, 6, 7`
- $523_8 = 5 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 339_{10}$

BASE 16

- Simboli: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`
- $3A2_{16} = 3 \cdot 16^2 + A \cdot 16^1 + 2 \cdot 16^0 = 930_{10}$

BASE 2

- Simboli: `0, 1`
- $101101_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 45_{10}$

9.1.2 Passare in base 2

Per passare da una base, ad esempio `10`, in base `2`, si può utilizzare il seguente metodo[1], che è il procedimento inverso di quelli mostrati nel paragrafo 9.1.1 nella pagina precedente.

S'individua la massima potenza di `2` che non supera il numero m di partenza (sia essa n). Se $2^n - m = z_1 = 0$, il numero binario si ottiene ponendo 1 nella posizione $n+1$ e zero nelle posizioni da n in giù. In caso contrario ($z_1 \neq 0$), si trova la massima potenza di `2` che non superi z_1 e si ripete il procedimento finché non s'individua un indice $i \in \{1, \dots, \log_2 n + 1\} \cap \mathbb{N} \mid z_i = 0$. Ad esempio:

$$45_{10} = \underbrace{32+8+4+1}_{2^5+2^3+2^2+2^0} = 101101_2$$

Tale algoritmo, tuttavia, è piuttosto scomodo da stendere come programma e risulterebbe "pesante" per il calcolatore. Si può allora pensare ad un altro procedimento.

Si divide il numero m di partenza per `2`. Il resto r_1 di tale operazione è la prima cifra (quella con il peso minore) del numero binario. Si divide ora il quoziente q_1 (intero) per `2`. Come prima, il resto è la seconda cifra del numero. Il procedimento termina quando s'individua un indice $i \in \{1, \dots, \log_2 n + 1\} \cap \mathbb{N} \mid q_i = 0$. Ecco un esempio:

$$\frac{45}{2} = 22 + \mathbf{1};\ \frac{22}{2} = 11 + \mathbf{0};\ \frac{11}{2} = 5 + \mathbf{1};\ \frac{5}{2} = 2 + \mathbf{1};\ \frac{2}{2} = 1 + \mathbf{0};\ \frac{1}{2} = 0 + \mathbf{1}.$$

Chiaramente, tale algoritmo è molto più semplice da tradurre in un programma in C e risulta molto più efficiente del precedente.

9.1.3 `bit` e `byte`

L'unità di misura della memoria in un calcolatore è il `bit` (*Binary Digit*, cioè *Cifra Binaria*). Il suo primo multiplo è il `byte`[2] (`1 byte = 8 bit`). Con un `byte` si ottengono $2^8 = 256$ combinazioni differenti, quindi, in teoria, 256 caratteri diversi. In realtà, la combinazione `00000000` non viene considerata un carattere, pertanto le rappresentazioni effettivamente ammesse sono $2^8 - 1 = 255$ (i caratteri ASCII). In generale, n `bit` danno 2^n combinazioni e $2^n - 1$ rappresentazioni di caratteri.

Nel calcolatore, normalmente, una variabile di tipo `int` occupa `4 byte` ossia $(2^8)^4 = 2^{8\cdot4} = 4294967296$ combinazioni. Se gli interi fossero soltanto positivi, il più grande numero rappresentabile sarebbe $4294967296 - 1 = 4294967295$. Tuttavia, variabili di tipo `int` possono assumere valori sia positivi che negativi. Si stabilisce, allora, che il primo `bit` di un intero sia destinato a contenere il segno del numero: per convenzione, `0 = "-"` e `1 = "+"`.

Si ammetta, per semplicità, che un intero sia rappresentato da `4 bit`. Allora, le rappresentazioni `1000` e `0000` corrispondono ($-0_{10} = +0_{10}$). Avendo destinato un `bit` al segno, restano soltanto $2^{\textit{numero di }\texttt{bit}} - 1$ rappresentazioni differenti, 255 nella fattispecie. Per ovviare a questo problema, si ricorre al metodo del *complemento a due*.

9.1.4 Complemento a due

S'immagini di sistemare sul bordo di un anello tutte le combinazioni date da un certo numero di `bit` (se ne considerino sempre `4`, per comodità) come

1 Si noti che per passare ad una base b qualsiasi, si può usare lo stesso procedimento sostituendo a 2 il numero b.

2 Corrispondente al "quadretto" nel paragrafo 2.2 a pagina 6

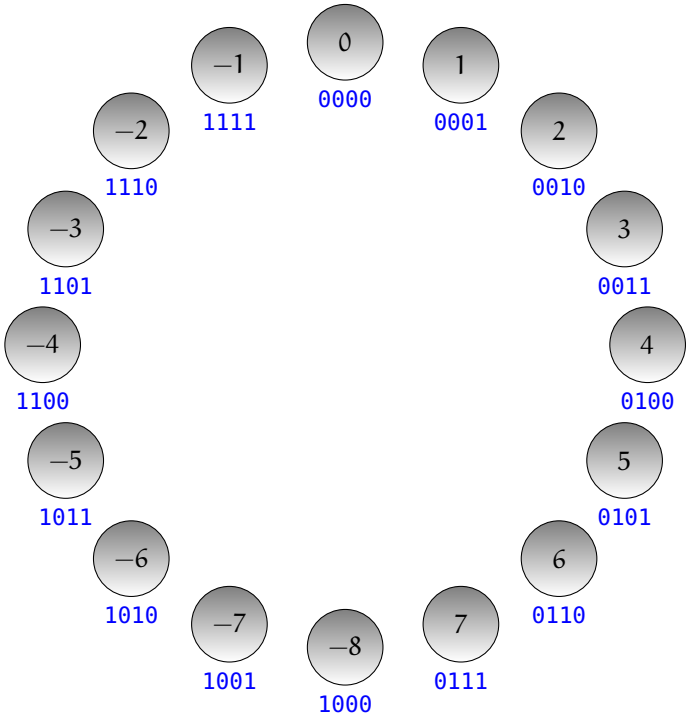


Figura 9.1: Rappresentazione del complemento a due con numeri di `4 bit`.

in figura 9.1. Ora, s'interpretino nel modo usuale i numeri che hanno come prima cifra (partendo da sinistra) `0`. Cominciando da `0000` e spostandosi in senso orario, dunque, si hanno numeri interi crescenti fino al numero `1000`. Si stabilisce, altresì, di assegnare ai numeri che hanno `1` come prima cifra valori progressivamente più piccoli partendo da -1 (`1111`) e proseguendo in senso antiorario.

Si ha che, ad esempio, considerando la somma delle rappresentazioni binarie dei numeri -6_{10} e -2_{10}:

```
  1010  +
  1110  =
-------
 11000
```

Siccome un numero è composto di `4 bit`, il calcolatore memorizzerà solo `1000` che corrisponde a `-8`. Si prendano altri casi:

$$3 + (-5) = \begin{array}{r} \texttt{0011} \;+ \\ \texttt{1011} \;= \\ \hline \texttt{1110} \end{array} \quad \rightarrow -2_{10}$$

Si noti che non c'è stato bisogno d'introdurre l'operazione di differenza. Una volta definita la somma, non serve altro. Ancora:

$$5 + (-3) = \begin{array}{r} \texttt{0101} \;+ \\ \texttt{1011} \;= \\ \hline \texttt{10010} \end{array} \quad \rightarrow 2_{10}$$

Tuttavia:

$$3 + 6 = \begin{array}{r} \texttt{0111} \;+ \\ \texttt{0110} \;= \\ \hline \texttt{1101} \end{array} \quad \rightarrow -3_{10}$$

Ciò accade perché il numero `13` non può essere rappresentato con `4 bit`. Infatti, tale metodo è corretto solo per numeri $n \in \{-8, \ldots, 7\} \cap \mathbb{N}$ (o, in generale, essendo m il numero di `bit`: $-2^{m-1} - 1 \leqslant n \leqslant 2^{m-1} - 1$). Questo è un errore comunemente chiamato *overflow*.

9.1.5 Virgola mobile

Finora, è stata trattata soltanto la rappresentazione di un intero ($n \in \mathbb{Z}$) in un calcolatore. Si considerino ora dei numeri $q \in \mathbb{Q}$. Affinché essi siano rappresentati in memoria si deve introdurre il concetto di *virgola mobile* (o, in inglese, *floating point*). Siano dati i seguenti numeri decimali:

$$2,718_{10} = 2 \cdot 10^0 + 7 \cdot 10^{-1} + 1 \cdot 10^{-2} + 8 \cdot 10^{-3};$$
$$101,01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}.$$

Nella sua rappresentazione più semplificata, un numero in virgola mobile è composto da:

- Mantissa;
- Esponente;
- Base.

Bisogna tenere conto che, come in precedenza, anche qui `1 bit` viene riservato al il segno. Le cifre significative sono rappresentate dalla *mantissa*. Si ha che:

$$2,718_{10} = \underbrace{2718}_{\textit{Mantissa}} \cdot \underbrace{10}_{\textit{Base}}{}^{(-3)} \rightarrow \textit{Esponente}$$

Tra le infinite rappresentazioni in virgola mobile, si stabilisce una forma "canonica". Essa prevede che:

- Siano m, b rispettivamente la mantissa e la base, allora si deve avere che: $0 < m < b$;
- L'esponente della base dev'essere il più piccolo possibile.

In memoria, un numero in virgola mobile è rappresentato secondo il seguente schema:

segno	*esponente*	*mantissa*

Se un numero in virgola mobile occupa `32 bit`, a segno, esponente e mantissa vengono assegnati rispettivamente: `1`, `8` e `23 bit`. Se ne occupa `64`, invece: `1`, `11` e `52 bit`.

Si tenga presente che nelle operazioni algebriche, il calcolatore interpreta la mantissa m come se fosse $m + 1$.

9.2 STRUTTURE DI CONTROLLO

Il ciclo `for()`

Durante tutta la trattazione è stata introdotta una sola notazione per le iterazioni: il ciclo `while()` (vedi il paragrafo 3.0.3 a pagina 11). Esiste, tuttavia, un altra sintassi che risulta in parecchie occasioni più conveniente: il ciclo `for()`.

Riquadro 4 Confronto tra il ciclo `while()` ed il ciclo `for()`.

```
int i = 0;
while ( i < N ) {
    s = s+p[i];
    i++;
}
```

```
int i;
for ( i = 0; i < 10; i++ ) {
    s = s+p[i];
}
```

9.2.1 Ancora sulle iterazioni

Il riquadro 4 mostra un ciclo `while()` e il suo equivalente ciclo `for()`. Il secondo prevede una sintassi molto più contenuta e, nella maggior parte dei casi, rende il codice molto più leggibile.

La sintassi del ciclo `for` è:

`for( /*istruzione 1*/; /*condizione*/; /*istruzione 2*/).`

Il compilatore:

- Esegue `/*istruzione 1*/` (in genere, un assegnamento) una sola volta, all'inizio del ciclo;
- Verifica la `/*condizione*/`;
- Se la `/*condizione*/` è verificata, esegue il corpo del ciclo;
- Esegue `/*istruzione 2*/`;
- Verifica di nuovo la condizione;

  ⋮

10 LEZIONE X

26/05/2011

10.1 BREVE STORIA DELL'INFORMATICA

L'*informatica* è una disciplina che si fonda su due pilastri fondamentali:

TEORIA DELLA COMPUTAZIONE; (Turing, Church, Post nel 1936) che si occupa di studiare il concetto di "algoritmo" e tutto ciò che ne consegue;

TEORIA DELL'INFORMAZIONE; (Shannon nel 1949) che nasce essenzialmente dal problema di trasportare messaggi da un luogo ad un altro nel modo più sicuro ed efficiente possibile.

10.1.1 Teoria della computazione

La *teoria della computazione* nasce dalla domanda: cosa è possibile calcolare meccanicamente?

Parte della teoria si spende nello specificare innanzitutto il significato della parola "meccanicamente". Data la generalità di questa trattazione, si può assumere che significhi "seguendo delle istruzioni fisse".

10.1.2 La macchina di Turing

Computer e contabili

Negli anni '30 esistevano già le calcolatrici, ma la tecnologia non rendeva ancora disponibile alcunché di più avanzato. C'erano, tuttavia i *computer* che, in lingua inglese, erano i contabili. Alan Mathison Turing (vedi il paragrafo 12.2.1 a pagina 51) s'ispirò proprio al modo di lavorare del contabile per mettere a punto la sua idea si calcolatore. Il contabile, infatti, è chiamato ad eseguire dei calcoli applicando formule date a priori, una matita ed un foglio. Non ha bisogno di usare la creatività o l'inventiva. Svolge solo dei calcoli e, di tanto in tanto, si serve del foglio e della matita per appuntare dei risultati intermedi. Il contabile ha inoltre a disposizione una quantità finita di segni differenti da poter usare. Riesce a riconoscere ed interpretare solo un numero finito di simboli. Anche il foglio (a quadretti) di cui dispone ha una dimensione limitata. Inoltre, in ogni quadretto c'è lo spazio per uno ed un solo simbolo. Egli può anche trovarsi in un numero finito di quelli che Turing denominò "stati mentali", ossia delle "condizioni interne" che lo fanno reagire in modo diverso a degli stimoli provenienti dall'esterno.

Macchina di Turing

Partendo da quanto detto finora, Turing immaginò la cosiddetta *Macchina di Turing* (figura 10.1). Molto schematicamente, si tratta di una scatola che può trovarsi in un numero finito di stati interni differenti. Dalla scatola si protende un braccio che dà su un nastro scorrevole. Quest'ultimo contiene dei numeri, dei caratteri alfanumerici o niente. Tale nastro può essere allungato indefinitamente, pur restando di lunghezza finita. Il braccio è in grado sia di leggere dal na-

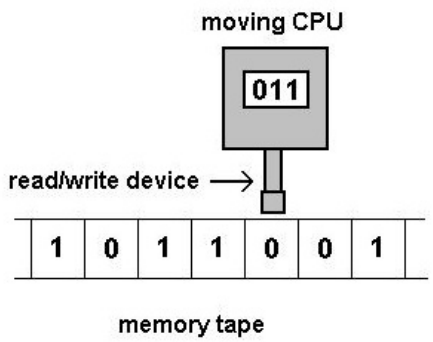


Figura 10.1: Macchina di Turing.

stro che di scrivervi sopra. Il cuore dell'apparecchio è la *tabella delle istruzioni*.

Sia S l'insieme degli stati interni assumibili dalla macchina e A l'insieme dei simboli che essa riesce a riconoscere:

$$\begin{cases} S = \{\sigma_1, \dots, \sigma_n\}; \\ A = \{\alpha_1, \dots, \alpha_m\}; \end{cases} \text{ con } m, n \in \mathbb{N}.$$

Le istruzioni della macchina, allora, hanno la forma:

> "se (il tuo stato è σ_i) e se (leggi α_e nella casella); allora (cambia stato in σ_r, sposta la testina a destra e scrivi...)"

Ad un certo punto, la macchina troverà un'istruzione di `stop` e si fermerà. Allora si possono fissare gli insiemi dell'alfabeto A e degli stati interni S (delle istruzioni):

- *Alfabeto*: codice binario;
- *Istruzioni:*
  - Se (la testina legge un numero) $\rightarrow$ (va a destra);
  - Se (la testina legge una casella vuota) $\rightarrow$ (scrive nella prima casella vuota "0").

In pratica, tale funzione aggiunge uno `0` ad un numero binario, che corrisponde a raddoppiarlo.

10.1.3 Problemi insolubili

Con questo tipo di macchine è possibile calcolare funzioni $f : \mathbb{N} \mapsto \mathbb{N}$ come, ad esempio, $f(n) = 2n$; $f(n) = \sqrt{n}$ (purché ci si accontenti della sola parte intera); $f(n) = n^2 + n + 1 \dots$ Esistono, tuttavia, delle funzioni che non possono essere calcolate con una macchina di Turing dal momento che non esiste un processo meccanico (algoritmo) che porti alla loro soluzione[1].

Polinomio di grado qualsiasi

Non esiste un algoritmo che, dato un polinomio a coefficienti interi di grado non fissato a priori, sia in grado di stabilire se esistano o meno delle radici intere. Tale risultato è ottenibile solo conoscendo in anticipo il grado del polinomio.

Problema dell'arresto

Un'altra questione non risolubile è il cosiddetto *problema dell'arresto*. Si consideri un codice del tipo:

```
i = 1;
while( i > 0 )
    i++;
```

Esso genera un ciclo teoricamente infinito. Una volta avviato non può arrestarsi, tranne che per cause esterne[2]. Dopo aver compilato il programma, non c'è alcuna possibilità d'individuare un errore di questo tipo durante l'esecuzione. Non si può dimostrare di aver scritto un ciclo infinito (anche se ad un certo punto, cominciano a sorgere forti sospetti...). Si potrebbe pensare, allora, di scrivere un programma che verifichi il codice di un altro dal punto di vista logico. Lo stesso Turing dimostrò che un siffatto programma non può essere steso.

1 Questo si può dimostrare provando che l'insieme funzioni aritmetiche calcolabili è un sottoinsieme di quello delle funzioni aritmetiche. Dal momento che una funzione aritmetica è calcolabile se e solo se esiste un algoritmo che la calcola [7], segue la tesi.

2 Vedi: "black out", oppure "utente incavolato che sfascia il computer perché il suo programmino non funziona"...

(a) *Rappresentazione binaria delle istruzioni.*

Istruzione	*Numero*
Sposta la testina a destra	`0`
Sposta la testina a sinistra	`1`

(b) *Rappresentazione binaria degli stati interni.*

Stato	*Numero*
σ_1	`01`
σ_2	`10`
σ_3	`11`

Tabella 10.1: Convenzioni binarie per la macchina di Turing

10.1.4 La macchina universale

Presa una macchina di Turing tale che:

$$\begin{cases} S = \{\sigma_1, \sigma_2, \sigma_3\}; \\ A = \{1, 0\}; \end{cases}$$
$$\sigma_1, 0 \to 1, \sigma_2, \text{ destra};$$
$$\vdots$$

Si possono tradurre gli stati interni del computer usando il suo alfabeto (il codice binario). Nello specifico, ci sono tre (11) stati , e due (10) simboli. Adottando le convenzioni delle tabelle 10.0a e 10.0b è possibile tradurre le istruzioni in linguaggio binario. Si ha che:

$$\begin{cases} \texttt{01 10 11} \\ \texttt{1 0} \end{cases}$$
$$\texttt{1\_0\_1\_10\_0}$$
$$\vdots$$

Il problema di distinguere i numeri da elaborare dalle istruzioni è facilmente risolvibile.

Ora, una macchina di Turing può essere descritta con un numero. Si consideri, allora, una macchina U che riesca a riconoscere parti di codice di altre macchine. Si hanno due macchine E e D tali che:

$$\begin{cases} E: f(x) = 2 \\ D: f(x) = x^2 - 1 \end{cases}$$

Se alla macchina E corrisponde il codice `1_0_11_0_1`, allora la macchina U riesce ad interpretarlo come istruzione ed applicarlo su un numero x (`101`, ad esempio). Tale macchina U è una cosiddetta *macchina Universale*. In questo modo non si deve progettare un apparecchio specifico per ogni algoritmo da eseguire, ma ne basta uno solo.

La tesi di Church-Turing

Da quanto detto finora, discende la *tesi*[3] *di Church-Turing*:

> «Le funzioni calcolabili dalla macchina di Turing sono *tutte e sole* quelle calcolabili meccanicamente.»

Gli studi di John von Neumann negli anni '40 (vedi il paragrafo 2.1 a pagina 5) si basavano proprio sulle osservazioni di Turing.

3 Si noti che *non* è un teorema perché discende direttamente da tutta la teoria "costruita" da Turing.

10.2 ALGORITMO DI DIJKSTRA PER I CAMMINI MINIMI

Un *grafo* (vedi il paragrafo 6.1.1 a pagina 23) è un oggetto composto di vertici (o *nodi*) e lati (o *archi*). Un arco "unisce" due nodi. Siano V ed E rispettivamente gli insiemi dei vertici e lati, allora:

$$V = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$
$$E \subseteq V \times V$$

Se si associa ad ogni lato $e_i \in E$ un numero $z_i \in \mathbb{R}^+$, è naturale interpretare z_i come la distanza che intercorre tra i due nodi collegati dall'arco e_i.Dato un grafo, è lecito chiedersi quale sia il *cammino minimo* (o cammino di minimo costo) tra due suoi punti qualsiasi. Per rispondere a tale domanda, ci si serve dell'*algoritmo di Dijkstra*.

Si vuole calcolare il cammino più breve che collega il vertice A della figura 10.2a con tutti gli altri. Per fare ciò è conveniente registrare tutti i possibili percorsi (con le rispettive distanze) in una tabella come la 10.2b. Si trovano i nodi immediatamente vicini ad A. Si ha che $\overline{AB} = 5$ e $\overline{AE} = 1$. Per ora, queste sono le distanze minime. Partendo ora da B, si ha che $\overline{AC} = \overline{AB} + \overline{BC} = 8$ e $\overline{AD} = \overline{AB} + \overline{BC} = 6$. Entrambi questi percorsi individuano, al momento, dei cammini minimi. Ripetendo l'operazione per tutti i vertici del grafo, si ottiene tabulata la distanza minima tra A e qualsiasi altro nodo (ma anche tra due vertici qualsiasi $v_1, v_2 \in \{A, B, C, D, E, F, G\}$).

Rappresentazioni in C

Per tradurre tale algoritmo in linguaggio C, c'è innanzitutto bisogno di trovare una rappresentazione per il grafo. Vi sono diverse soluzioni:

- Si rappresentano i vertici con degli interi e il lati con un record che contenga informazioni sui nodi che collega e la loro distanza;
- Si usa una matrice.

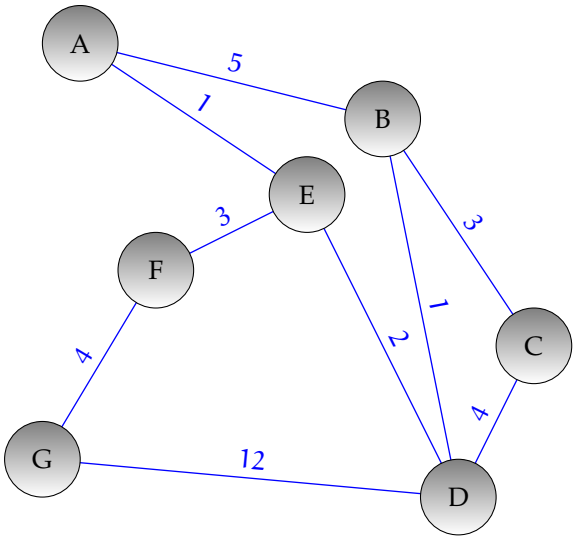
  Avendo dei vertici $v_{i \in \{1,\ldots,N\}} \in V$, si dichiara una matrice $N \times N$ come la tabella 10.2c nella pagina successiva e si stabilisce che:

  - Se tra due vertici c'è un lato, si scrive nella casella corrispondente il valore 1 (o la distanza $z \in \mathbb{R}^+$);
  - Se tra due vertici non c'è un lato, si scrive nella casella corrispondente il valore 0.

  Si noti, che avendo dichiarato una matrice `float m[N][N];` si ha che `m[e][j] == 0` $\forall e, j \in \{1, \ldots, N\} \mid e = j$ se non esistono archi che colleghino un punto con se stesso, come in figura 10.2a. Con questa rappresentazione, il costo computazionale dell'algoritmo di Dijkstra (siccome sulla diagonale ci sono solo valori nulli) è $N^2 - N \sim N^2$ per $N \to +\infty$. Pertanto, la rappresentazione matriciale risulta piuttosto scomoda se il grafo è sparso, cioè se ci sono pochi archi.
- Ci si serve di una *lista adiacenze*.

  Essa, come mostra la figura 10.2d a fronte, consiste in più serie di record ognuna delle quali si riferisce ad un nodo di partenza differente. Ogni struttura contiene un campo che tenga traccia del nome del nodo collegato e un campo puntatore alla struttura successiva. Nel campo puntatore dell'ultimo record, ormai è inutile dirlo, si pone il valore `NULL`.

  Tale rappresentazione risulta piuttosto ridondante per grafi non orientati perché si usano due record per ogni arco, ma diventa più efficiente nel caso di grafi orientati. Si noti che, volendo aggiungere il "peso" di ogni lato, si dovrebbe aggiungere ad ogni record un campo dedicato.



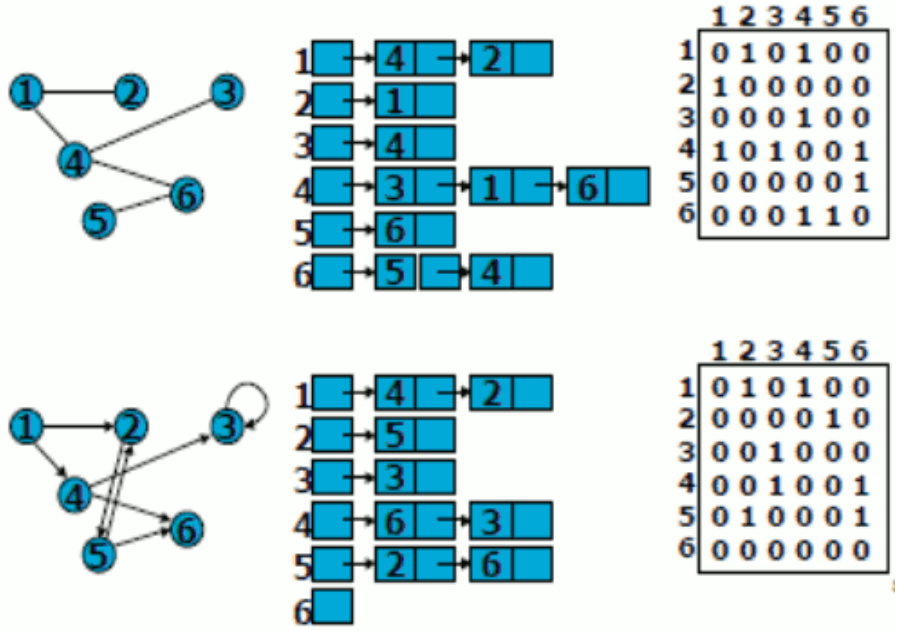
(a) *Rappresentazione del grafo.*

Vertice	*Distanza*	*Percorso*
A	0	/
B	**5**	**AB**
C	8; **7**	ABC; **AEDC**
⋮	⋮	⋮

(b) *La tabella contiene alcune distanze partendo dal vertice A. Il cammino minimo è evidenziato.*

	v_1	v_2	…	v_N
v_1	0	4.5	…	0
v_2	3	0	…	5.7
⋮	⋮	⋮	⋱	⋮
v_N	8.9	0	…	0

(c) *Matrice $n \times n$. Si noti che sulla diagonale ci siano solo valori nulli.*



(d) *Grafo con rappresentazione per lista adiacenze e rappresentazione matriciale.*

Figura 10.2: Algoritmo di Dijkstra

11 LEZIONE XI

06/03/2011

11.1 LEZIONE PRE-ESAME

Per un manuale affidabile sulla sintassi in C, è possibile fare riferimento alla voce `C syntax` su Wikipedia (in inglese).

11.1.1 Modello (o struttura) dei dati

Le strutture dati studiate finora sono le seguenti (tranne le ultime due, cui s'accennerà in seguito):

- Array o vettori, matrici (vedi il paragrafo 3.1.1 a pagina 12);
- Grafi (vedi il paragrafo 6.1.1 a pagina 23);
- Record o strutture (vedi il paragrafo 6.1.2 a pagina 23);
- Stack (vedi il paragrafo 7.2.1 a pagina 27);
- Liste concatenate (vedi il paragrafo 7.4.1 a pagina 30);
- Code (vedi il paragrafo 11.2.1);
- Alberi (vedi il paragrafo 11.2.2 nella pagina successiva).

Ciascuno di questi modelli si caratterizza per le operazioni che permette di compiere. Nelle liste concatenate trattate nel paragrafo 7.2.1 a pagina 27, ad esempio, sono definiti l'inserimento (con la funzione `push()`) e la cancellazione di elementi (tramite la funzione `pop()`).

In ogni caso, una struttura dati non è troppo vincolante: è possibile creare una lista concatenata usando una matrice, ad esempio. Dichiarando una matrice di n righe, si possono destinare le prime $n-1$ al "carico utile" e l'n-esima a puntare al nodo successivo. In parole povere, si usa l'n-esima riga come campo `*next`. Così facendo, è come se una colonna di una matrice rappresentasse un nodo. Tuttavia, poiché le matrici sono indicizzate tramite interi z progressivi, basterà un intero per identificare l'indirizzo del nodo successivo. Come ultimo accorgimento, basta stabilire che il nodo "di coda" contenga nella sua n-esima riga un intero negativo (come `-1`) ad indicare la fine della lista. Ad ogni modo, servirà comunque una variabile ausiliaria (analogamente al puntatore `*testa`) di tipo `int` che indichi qual è il primo nodo.

Per l'esame, potrebbe essere necessario apportare delle modifiche ad una struttura dati esistente, come creare una lista concatenata bidirezionale (vedi la figura 11.1d a pagina 47). Tale struttura ha bisogno di due campi puntatore (`*prev` e `*next`) e può essere percorsa in entrambi i versi. Oppure, si potrebbero dover dichiarare più campi puntatori all'interno di un record per ordinare la lista in base a criteri differenti.

11.2 STRUTTURE DATI

11.2.1 Coda

La *coda* (figura 11.1e a pagina 47) è una struttura dinamica lineare: gli elementi, cioè, sono ordinati. A differenza di uno stack, tuttavia, essi seguono una disciplina del tipo FIFO (*First In First Out*). Un nuovo oggetto inserito si colloca in fondo alla coda, mentre per estrarne uno lo si preleva dalla testa.

Tabella 11.1: La tabella mostra il costo computazionale di uno stack, una coda ed una coda con un puntatore all'ultimo elemento, con n elementi.

Struttura	*Inserimento*	*Eliminazione*
Stack	1	1
Coda	n	1
Coda (puntatore)	1	1

Risulta ora evidente l'analogia di tale struttura con una fila (o coda) reale che si fa per usufruire di un servizio ad uno sportello, da cui peraltro prende il nome. Differenze tra coda e stack si riscontrano anche nel *costo computazionale*, ossia sul numero di "calcoli" che l'esecutore deve compiere per effettuare un'operazione sulla struttura, come mostra la tabella 11.1. Per eliminare un elemento, partendo da `*testa`, il computer deve percorrere tutta la coda finché non giunge all'ultimo. Per "snellire" il costo computazionale della coda, è possibile introdurre un puntatore al nodo finale in modo tale che l'eliminazione comporti un solo passaggio, al pari dell'inserimento.

11.2.2 Albero

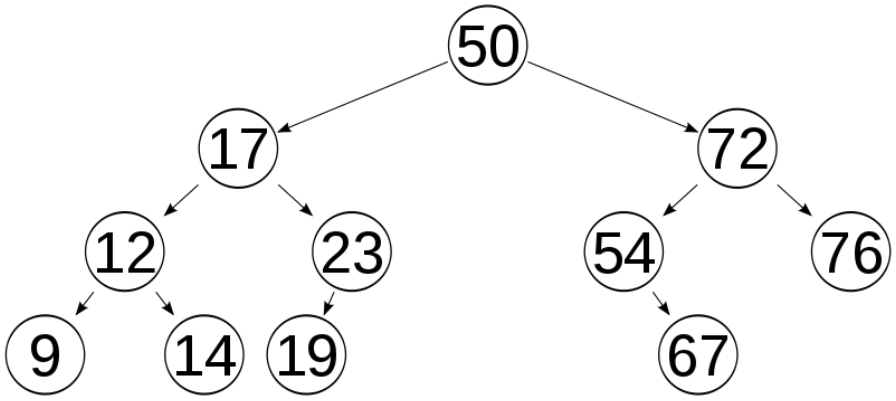
Un *albero* è un modello dei dati che sfrutta una struttura gerarchica. Esistono diverse implementazioni di albero; in prima battuta si può considerare una rappresentazione che preveda due campi puntatore per ogni struttura. Siano essi `*min` e `*max`.

Si supponga di aver bisogno di tenere ordinati n dati numerici inseriti nello `stdin`. All'inserimento del primo dato d_1 si crea un nodo (sia esso `n_1`), chiamato *capostipite* o *padre* (il "pallino" rosso in figura 11.1c a fronte). All'ingresso del secondo dato d_2 si crea un altro nodo (`n_2`) e si stabilisce che:

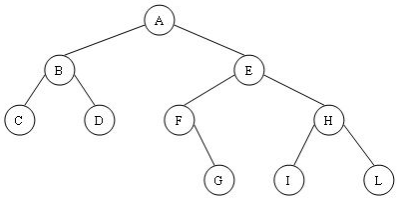
- $d_1 > d_2 \quad \rightarrow \quad$ `min = &n_2;`
- $d_1 < d_2 \quad \rightarrow \quad$ `max = &n_2.`

Ripetendo il procedimento n volte, si avranno dei record contenenti numeri ordinati (in ordine crescente, in questo caso).

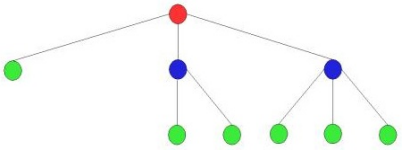
Per com'è stato costruito quest'albero, ogni *genitore* può essere collegato al massimo a due *figli*. Esso è, quindi, un albero binario o BST (*Binary Search Tree*, figura 11.1a nella pagina successiva). Ad ogni modo, è possibile creare una struttura "dal basso", in cui ogni albero figlio ha un puntatore al genitore. Tale modifica permette di avere un albero non più (soltanto) binario ma anche a più elementi.



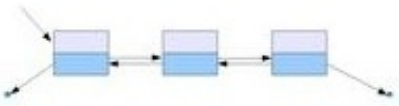
(a) *Binary Search Tree*



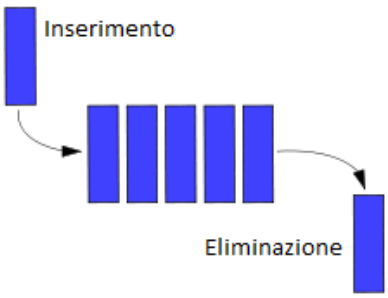
(b) *BST con caratteri alfabetici*



(c) *Rappresentazione grafica di un albero*



(d) *Lista concatenata bidirezionale. I campi in blu sono puntatori.*



(e) *Rappresentazione grafica di una coda.*

Figura 11.1: Alcune strutture dati.

12 DOMANDE E (ALCUNE) RISPOSTE D'ESAME

La maggior parte di questo e del prossimo capitolo è stata spudoratamente scopiazzata dalla mitica Viky Pedya (sempre sia lodata), unica e sola fonte di salvezza degli studenti in crisi (e non). Mi sarebbe piaciuto attingere anche da siti più competenti come Nonciclopedia, cui comunque rimando per questo articolo.

12.1 DOMANDE D'ESAME

BERNARDINELLO'S PREFACE Questo documento ha lo scopo di aiutarvi nella preparazione dell'esame. Per superare l'esame è necessario saper rispondere correttamente alle domande che seguono (necessario ma non sufficiente).

Chi ha seguito il corso dovrebbe essere in grado di rispondere alla maggior parte delle domande. Per le domande più nozionistiche, vi è richiesto un piccolo sforzo di ricerca e documentazione. A questo scopo, sfruttate la biblioteca di ateneo; l'Internet è un'altra utile fonte di informazioni, ma va considerata inaffidabile.

Se non vi sentite in grado di rispondere a qualcuna delle domande, stabilite innanzitutto se la questione non è stata trattata a lezione in modo adeguato; in tal caso, segnalatemelo immediatamente.

L'elenco viene di tanto in tanto aggiornato. Tornate a consultarlo.

Luca Bernardinello

FONDAMENTI E PRINCIPI DELL'INFORMATICA:

1. Chi era Alan Mathison Turing (vedi il paragrafo 12.2.1 a pagina 51)?
2. Che cos'è la Macchina di Turing? Che relazione ha con i calcolatori (vedi il paragrafo 10.1.2 a pagina 39)?
3. Chi era John von Neumann? Descrivere la struttura generale di un calcolatore secondo von Neumann (vedi il paragrafo 2.1 a pagina 5).
4. Che cos'è un algoritmo? Dare una definizione il più possibile rigorosa.

ARCHITETTURA DEI CALCOLATORI E SISTEMI OPERATIVI:

5. Descrivere la struttura di un calcolatore (vedi il paragrafo 2.1 a pagina 5).
6. Che cos'è un `bit`? Che cos'è un `byte` (vedi il paragrafo 12.2.2 a pagina 52)?
7. Che cos'è un Operating System (OS)? Quali sono le sue funzioni principali (vedi il paragrafo 12.2.3 a pagina 53)?
8. Come si interagisce con un OS (vedi il paragrafo 12.2.4 a pagina 54)?

ALGORITMICA E PROGRAMMAZIONE:

9. Che cosa si intende per *pseudocodice* (vedi il paragrafo 12.2.5 a pagina 55)?
10. Che cos'è un Linguaggio di Programmazione (vedi il paragrafo 12.2.6 a pagina 55)?
11. Che cosa si intende per Strutture Dati? Elencate e descrivete qualche struttura di dati dinamica (vedi il paragrafo 12.2.7 a pagina 56).
12. Quali sono le Strutture di Controllo necessarie per esprimere un algoritmo (vedi il paragrafo 2.4 a pagina 7)?
13. Che cosa si intende per *costo computazionale* di un algoritmo? Come si calcola?
14. L'algoritmo di ordinamento *merge sort* ha un costo computazionale asintotico a $O(n \log_2 n)$. Che cosa vuol dire (vedi il paragrafo 1.1.1 a pagina 2)? (a proposito: che cos'è un algoritmo di ordinamento?)
15. Che differenza c'è fra costo computazionale di un algoritmo e costo computazionale di un problema?
16. Che cosa si intende per correttezza di un algoritmo? Come possiamo verificare se un algoritmo è corretto?

12.2 (ALCUNE) RISPOSTE D'ESAME

12.2.1 Chi era Alan Mathison Turing?

Introduzione

Figura 12.1: Turing in una foto risalente al 29 Marzo 1951.

Alan Mathison Turing (Londra, 23 giugno 1912 – Wilmslow, 7 giugno 1954) è stato un matematico, logico e crittanalista britannico, considerato uno dei padri dell'informatica e uno dei più grandi matematici del Novecento. Introdusse la macchina ideale e fu anche uno dei più brillanti decrittatori che operavano in Inghilterra durante la seconda guerra mondiale per decifrare i messaggi scambiati da diplomatici e militari delle Potenze dell'Asse.

Omosessuale, morì suicida a soli 42 anni in seguito ad una persecuzione omofobica condotta nei suoi confronti. In suo onore la Association For Computing Machinery (ACM) ha creato nel 1966 il "Turing Award", massima riconoscenza nel campo dell'informatica, dei sistemi intelligenti e dell'intelligenza artificiale.

Biografia di Turing

Nascita

Turing venne concepito in India, durante uno dei viaggi di suo padre, Julius Mathison Turing. Sia Julius che sua moglie, Ethel Sara Stoney, madre del futuro Alan Turing, decisero che il piccolo dovesse nascere sul suolo inglese. Tornarono quindi a Londra dove il 23 Giugno 1912 nacque Alan (figura 12.1).

Infanzia e giovinezza

Già fin dalla più tenera età, Turing diede segno della genialità. Tuttavia, a causa della sua passione per le materie scientifiche divenne inviso ai professori del St. Michael, la sua prima scuola, i quali avevano sempre posto più enfasi sugli studi classici. Durante i primi anni di scuola ebbe quindi grosse difficoltà, ottenendo a stento il diploma. Poco appassionato al latino e alla religione, preferiva letture riguardanti la teoria della Relatività, i calcoli astronomici, la chimica o il gioco degli scacchi.

Gli anni dell'Università

Nel 1931 venne ammesso al King's College dell'Università di Cambridge dove approfondì i suoi studi sulla meccanica quantistica, la logica e la teoria della probabilità (dimostrò separatamente il *Teorema Del Limite Centrale*, già dimostrato nel 1922 da Lindeberg). Nel 1934 si laureò con il massimo dei voti e nel 1936 vinse il Premio Smith (riconoscimento che veniva assegnato ai due migliori studenti ricercatori in Fisica e Matematica presso l'Università di Cambridge). Nello stesso anno si trasferì alla Princeton University dove studiò per due anni, ottenendo infine un Ph.D. In questi anni pubblicò l'articolo "On computable Number, with an application to the Entscheidung-

sproblem" dove descriveva, per la prima volta, quella che sarebbe poi stata definita come la *macchina di Turing*.

Il lavoro come crittografo

Durante la seconda guerra mondiale, Turing mise le sue capacità matematiche al servizio del *Department of Communications* inglese per decifrare i codici usati nelle comunicazioni tedesche. Con l'entrata in guerra dell'Inghilterra Turing fu "arruolato" nel gruppo di crittografi stabilitosi a Bletchley Park e con i suoi compagni lavorò stabilmente. Fu sul concetto di macchina di Turing che nel 1942 il matematico Max Newman progettò una macchina chiamata *Colossus* (antesignana dei computer) che decifrava in modo veloce ed efficiente i codici tedeschi.

Il dopoguerra

Al termine della guerra Turing fu invitato al National Physical Laboratory (NPL) a Londra per disegnare il modello di un computer. Il suo rapporto, che proponeva l'Automatic Computing Engine (ACE), fu presentato nel marzo 1946 ma ebbe scarso successo a causa degli alti costi preventivati. Per l'anno accademico 1947/48 tornò a Cambridge e spostò i suoi interessi verso la neurologia e la fisiologia. Fu in questo periodo che iniziò ad esplorare la relazione tra i computer e la natura.

Test di Turing

Nel 1950 scrisse un articolo dal titolo "Computing Machinery And Intelligence" sulla rivista Mind in cui descriveva quello che sarebbe divenuto noto come il *Test di Turing*. Su questo articolo si basa buona parte dei successivi studi sull'intelligenza artificiale. L'anno seguente fu eletto Membro della Royal Society di Londra. Si trasferì all'Università di Manchester, dove lavorò alla realizzazione del Manchester Automatical Digital Machine (MADAM). Convinto che entro l'anno 2000 sarebbero state create delle macchine in grado di replicare la mente umana, lavorò alacremente creando algoritmi e programmi per il MADAM, partecipò alla stesura del manuale operativo e ne divenne uno dei principali fruitori.

Reclusione e morte

Nel 1952 sviluppò un approccio matematico all'embriologia. Il 31 marzo dello stesso anno fu arrestato per omosessualità e condotto in giudizio, dove a sua difesa disse semplicemente che «non scorgeva niente di male nelle sue azioni». Secondo alcune fonti, Turing avrebbe denunciato per furto un suo amico ospite in casa sua ed ammesso la propria tendenza in risposta a delle domande pressanti della polizia. In quel periodo si dibatteva nel parlamento britannico l'abrogazione del reato di omosessualità e ciò probabilmente avrebbe indotto Turing ad un comportamento incauto. Fu sottoposto alla castrazione chimica, che lo rese impotente e gli causò lo sviluppo del seno; alcuni dei motivi che probabilmente lo condussero, di lì a poco, al suicidio. Nel 1954 Alan Turing morì ingerendo una mela avvelenata con cianuro di potassio, in tono col proprio carattere eccentrico e prendendo spunto dalla fiaba di Biancaneve da lui apprezzata fin da bambino. La madre sostenne che il figlio, con le dita sporche per qualche esperimento chimico, avesse ingerito per errore la dose fatale di veleno; ma il verdetto ufficiale parlò senza incertezze di suicidio:

> «Causa del decesso: cianuro di potassio autosomministrato in un momento di squilibrio mentale.»

12.2.2 Che cos'è un `bit`? Che cos'è un `byte`?

In informatica, la parola `bit` ha due significati molto diversi, a seconda del contesto in cui rispettivamente la si usa:

- É l'unità di misura dell'informazione (dall'inglese *binary unit*), definita come la quantità minima di informazione che serve a discernere tra due possibili alternative equiprobabili;

- É una *cifra binaria*, (in inglese *binary digit*) ovvero uno dei due simboli del sistema numerico binario, classicamente chiamati zero e uno (0 e 1).

Il `bit` come quantità d'informazione

Nella prima accezione, un `bit` rappresenta l'unità di misura della quantità d'informazione. Intuitivamente, equivale alla scelta tra due valori equiprobabili (sì/no, vero/falso...). Matematicamente, la quantità d'informazione in bit di un evento è l'opposto del logaritmo in base due della probabilità (sia essa p) di tale evento ($-\log_2 p$). La scelta del numero 2 come base del logaritmo è significativa nel caso elementare di scelta tra due alternative (informazione di un `bit`), ma è possibile usare anche e (numero di Nepero), usando dunque il logaritmo naturale ($-\ln p$). In tal caso l'unità di misura dell'informazione si dice `Nat`.

Nel caso di due eventi equiprobabili, ognuno ha probabilità $1/2 = 0,5$. La loro quantità di informazione è, quindi, $-\log_2(1/2) = 1$ `bit`.

Il `bit` come cifra binaria

Nella seconda eccezione, il `bit` rappresenta l'unità di definizione di uno stato logico. La rappresentazione logica del `bit` è rappresentata dai soli valori $\{0, 1\}$. Ai fini della programmazione è comune raggruppare sequenze di `bit` in entità più vaste. Questi raggruppamenti contengono generalmente un numero di stringhe binarie pari ad una potenza binaria, pari cioè a 2^n con $n \in \mathbb{N}$. Il più noto è il `byte` (chiamato anche *ottetto*), corrispondente ad 8 `bit`, che costituisce l'unità di misura più utilizzata in campo informatico.

Il `byte`

Altre unità di Misura

Altri raggruppamenti di questo tipo sono i seguenti:

NIBBLE: 4 `bit` (1/2 `byte`);

WORD: di lunghezza variabile, corrisponde a 16, 32 o 64 `bit`;

DOUBLE WORD: pari a 2 `word` (`DWORD` o `LONGWORD`);

QUAD WORD: pari a 4 `word` (`QWORD`);

KIBIBYTE: 1024 `byte`, indicato con `KiB`;

MEBIBYTE: 1024 `kibibyte`, indicato con `MiB`;

GIBIBYTE: 1024 `mebibyte`, indicato con `GiB`;

TEBIBYTE: 1024 `gibibyte`, indicato con `TiB`;

PEBIBYTE: 1024 `tebibyte`, indicato con `PiB`;

EXBIBYTE: 1024 `pebibyte`, indicato con `EiB`;

ZEBIBYTE: 1024 `exbibyte`, indicato con `ZiB`;

YOBIBYTE: 1024 `zebibyte`, indicato con `YiB`.

12.2.3 Che cos'è un OS? Quali sono le sue funzioni principali?

Definizione

I'OS è un particolare software, installato su un sistema di elaborazione, senza il quale non è possibile l'utilizzo di altri software più specifici (in ultimo, del computer stesso). Esso quindi funge da "base" al quale si appoggiano gli altri software, che dovranno essere progettati in modo da essere riconosciuti e supportati da quel particolare sistema operativo. Per OS s'intende dunque l'insieme dei componenti software che hanno il duplice scopo di gestire le risorse hardware e software del computer, ed interfacciare l'utente con l'hardware.

Funzioni Principali di un OS

Secondo una definizione più rigorosa, il sistema operativo è un insieme di subroutine e strutture dati responsabili di:

- Controllo e della gestione delle componenti hardware che costituiscono il computer (processi di Input/Output da/verso le periferiche collegate al sistema);

- Esecuzione dei programmi che su di esso vengono eseguiti.

Se il sistema di elaborazione prevede la possibilità di memorizzazione aggiuntiva dei dati su memoria di massa, ha anche il compito di gestire l'archiviazione e l'accesso ai file. I programmi possono gestire l'archiviazione dei dati su memoria di massa (ottenendo strutture complesse, come un database), servendosi delle procedure messe a disposizione del sistema operativo. La componente dell'OS che si occupa di tutto ciò viene chiamata *File System*.

Infine, se è prevista interazione con l'utente, viene solitamente utilizzata allo scopo un'interfaccia software (grafica o testuale) per accedere alle risorse hardware del sistema. Solitamente un OS installato su computer fornisce anche degli applicativi di base per svolgere elaborazioni di diverso tipo.

Al di là delle prestazioni massime offerte dall'hardware dell'elaboratore stesso, l'OS determina di fatto efficienza e buona parte delle prestazioni effettive di funzionamento dell'intero sistema ad esempio in termini di latenze di processamento, stabilità, interruzioni o crash di sistema.

Struttura

Un generico sistema operativo moderno si compone di alcune parti standard:

- *Kernel*: gruppo di funzioni fondamentali, interconnesse fra loro e con l'hardware, che vengono eseguite con il privilegio massimo disponibile sulla macchina. Il kernel fornisce le funzionalità di base per tutte le altre componenti del sistema operativo, che assolvono le loro funzioni servendosi dei servizi che esso offre;

- *Gestore di File System*: si occupa di esaudire le richieste di accesso alle memorie di massa. Viene utilizzato ogni volta che si accede a un file su disco, e oltre a fornire i dati richiesti tiene traccia dei file aperti, dei permessi di accesso ai file. Inoltre si occupa anche e soprattutto dell'astrazione logica dei dati memorizzati sul computer (directory, ecc);

- *Sistema di Memoria Virtuale*: alloca la memoria richiesta dai programmi e dal sistema operativo stesso, salva sulla memoria di massa le zone di memoria temporaneamente non usate dai programmi e garantisce che le pagine swappate vengano riportate in memoria se richieste;

- *Scheduler*: che scandisce il tempo di esecuzione dei vari processi e assicura che ciascuno di essi venga eseguito per il tempo richiesto;

- *Spooler*: riceve dai programmi i dati da stampare e li stampa in successione, permettendo ai programmi di proseguire senza dover attendere la fine del processo di stampa;

- *Interfaccia utente* (Shell): permette agli esseri umani di interagire con la macchina.

12.2.4 Come si interagisce con un OS?

Si distinguono due forme d'interazione con un OS:

1. Interazione gestuale;

2. Interazione verbale.

Interazione Gestuale

Si parla d'*interazione gestuale* quando l'utente impartisce comandi al sistema operativo compiendo gesti che sfruttano dispositivi fisici (mouse, tastiera...) ed elementi visivi (figurine, menu, finestre...). Questa è la forma oggi largamente più diffusa.

Interazione Verbale

Si ha, invece, *interazione verbale* quando i comandi sono impartiti nel corso di un dialogo, sotto forma di parole o sigle. In questo caso l'interazione lascia una traccia su una console, che può occupare l'intero schermo oppure, ed è oggi il caso più frequente, una finestra sullo schermo. Le due forme di interazione possono quindi mescolarsi, e possiamo avere una sessione di interazione verbale nel contesto di un'interazione gestuale. Una sessione di interazione verbale è governata da un processo, quindi dall'esecuzione di un particolare programma, detto *shell* (cioè "guscio"). Nei sistemi operativi della famiglia Windows, questo guscio è denominato *prompt dei comandi*. Nei sistemi UNIX, e nei suoi derivati, esistono numerose varianti di shell. Di norma, essa segnala il proprio stato di attesa di comando attraverso un invito (*prompt*, appunto), cioè una particolare sequenza di caratteri visualizzata all'inizio della riga corrente nella console. La forma dell'invito varia da sistema a sistema, e può essere personalizzata da ogni singolo utente.

Shell

12.2.5 Che cosa si intende per *pseudocodice*?

Per *pseudocodice* (*pseudolinguaggio* o *linguaggio di progetto*) si intende un linguaggio di programmazione fittizio, non direttamente compilabile o interpretabile da un programma compilatore o interprete, il cui scopo è quello di rappresentare algoritmi. Lo pseudolinguaggio può essere utilizzato alternativamente al diagramma di flusso (*flow cahrt*). Lo pseudolinguaggio dipende dal paradigma di programmazione scelto, mentre dovrebbe essere pressoché indipendente dal linguaggio di programmazione, purché quest'ultimo rispetti naturalmente il paradigma scelto.

12.2.6 Che cos'è un Linguaggio di Programmazione?

Definizione

Un *linguaggio di programmazione* è un linguaggio formale, dotato di un lessico, di una sintassi e di una semantica ben definiti. È utilizzabile per il controllo del comportamento o la programmazione di una macchina formale attraverso la scrittura di un programma sotto forma di codice. Tutti i linguaggi di programmazione possiedono:

Elementi "obbligatori"

- *Variabile*: un dato o un insieme di dati, noti o ignoti, già memorizzati o da memorizzare; ad una variabile corrisponde sempre, da qualche parte, un certo numero (fisso o variabile) di locazioni di memoria che vengono allocate per contenere i dati stessi. Molti linguaggi attribuiscono alle variabili un tipo, con differenti proprietà;
- *Istruzione*: un comando o una regola descrittiva. Ogni volta che un'istruzione viene eseguita, lo stato interno del calcolatore cambia.

Concetti frequenti

Alcuni concetti sono poi presenti nella gran parte dei linguaggi:

- *Espressione*: una combinazione di variabili e costanti, unite da operatori. Una espressione viene valutata per produrre un valore;
- *Strutture Dati*: meccanismi che permettono di organizzare e gestire dati complessi;
- *Strutture di Controllo*: permettono di governare il flusso d'esecuzione del programma, alterandolo in base al risultato o valutazione di una espressione;

- *Sottoprogramma* (funzione): un blocco di codice che può essere richiamato da qualsiasi altro punto del programma;
- Funzionalità di input di dati da tastiera e visualizzazione dati in output (stampa a video).

12.2.7 Che cosa si intende per Strutture Dati? Elencate e descrivete qualche struttura di dati dinamica.

Una *struttura dati* è un'entità usata per organizzare un insieme di dati all'interno della memoria del computer. La scelta delle strutture dati è legata a quella degli algoritmi. Tale scelta, infatti, influisce inevitabilmente sull'efficienza degli algoritmi che la manipolano. La struttura dati è un metodo di organizzazione dei dati, quindi prescinde dai dati effettivamente contenuti.

I linguaggi di programmazione forniscono un insieme predefinito di tipi di dato elementari, e le strutture dati sono strumenti per costruire tipi di dati aggregati più complessi. Esse si differenziano soprattutto in base alle operazioni che si possono effettuare su di esse e alle prestazioni offerte.

Costruttori di Strutture Dati

Array (o Vettore) *[vedi il paragrafo 3.1.1 a pagina 12]*

Un *Array* è una struttura dati omogenea, che contiene un numero n finito di elementi dello stesso tipo. Questi elementi sono individuati attraverso un indice numerico (da 0 a $n-1$). La dimensione del vettore deve essere dichiarata al momento della sua creazione.

Record (o Struttura) *[vedi il paragrafo 6.1.2 a pagina 23]*

Un *Record* è una Struttura Dati che può essere eterogenea o omogenea. Nel primo caso contiene una combinazione di elementi che possono essere di diverso tipo. Questi sono detti *campi*, e sono identificati da un nome.

Strutture Dati dinamiche

Le Strutture Dati *dinamiche* sono basate sull'uso di dati di tipo *puntatore* e sull'allocazione dinamica della memoria. Lo spazio di memoria necessario per allocare i puntatori, e le operazioni necessarie alla loro manutenzione costituiscono il *costo aggiuntivo* delle strutture dati dinamiche.

Lista Concatenata *[vedi il paragrafo 7.4.1 a pagina 30]*

Una *Lista Concatenata* è un insieme di *nodi* collegati linearmente. I nodi sono dei record che contengono un "carico utile" di dati, ed un puntatore all'elemento successivo della lista. L'ordine con cui sono collegati i nodi definisce un ordinamento tra di loro. Un nodo funge da testa della lista, e da questo è possibile accedere a tutti i nodi della lista. Conoscendo un nodo interno alla lista, è possibile accedere ai nodi successivi, ma non a quelli precedenti. Il costo di accesso ad un nodo della lista cresce con la dimensione della lista. Conoscendo il nodo precedente ad un nodo N, è possibile rimuovere N dalla lista, o inserire un elemento prima di lui, in un tempo costante.

Lista Bidirezionale

Se la Lista è *bidirezionale*, ogni nodo contiene un puntatore sia al precedente che al successivo. Usando la sintassi del linguaggio C, dato un nodo N il suo successore è `N->succ`, e il suo precedente è `N->prec`. Deve sempre essere vero che `N->succ->prec == N`. Ogni nodo permette di accedere a tutti gli elementi della lista. Gli elementi "strutturali" di questa Struttura Dati, ovvero i due puntatori contenuti in ogni nodo, sono ridondanti

Alberi [vedi il paragrafo 11.2.2 a pagina 46]

Un *albero* è una rappresentazione dell'albero in teoria dei grafi. Ogni nodo contiene dei puntatori ad altri nodi che sono detti suoi *figli*. Dato un nodo, è possibile accedere a tutti i suoi discendenti. Non deve essere possibile partire da un nodo, seguire i puntatori ai figli e ritornare al nodo di partenza. Inoltre ciascun nodo deve essere figlio di un solo *padre*.

In alcune implementazioni, ogni nodo contiene un collegamento al suo padre. Tra i figli di un nodo esiste normalmente una relazione d'ordine,

definita dall'ordine dei puntatori nel padre. In molte implementazioni, ogni nodo ha un numero fissato di figli, ad esempio due o tre. Si parla in questo caso di alberi *binari* o *ternari*. In altri casi, il numero di figli di un nodo è arbitrario.

Binary Search Tree

Gli alberi binari si prestano anche a rappresentare insiemi dotati di ordinamento: un nodo contiene un elemento dell'insieme da ordinare; tutti gli elementi collegati al primo figlio sono precedenti; quelli collegati al secondo sono successivi. Questi alberi vengono anche definiti *Binary Search Tree* (BST) ovvero alberi di ricerca binaria. In questo modo, la ricerca di un elemento in un albero ordinato equilibrato richiede un tempo proporzionale al logaritmo del numero di elementi. L'inserimento di un elemento in un albero ordinato richiede di rispettare l'ordinamento precedentemente descritto.

Contenitori

Le strutture dati sopra esposti possono essere utilizzate per realizzare alcuni tipi di *contenitori* di utilizzo frequente, che possono forzare una particolare modalità di accesso ai dati.

Stack (o Pila) [vedi il paragrafo 7.2.1 a pagina 27]

Il termine *stack* (o pila) viene usato per riferirsi a strutture dati le cui modalità d'accesso seguono una politica LIFO (*Last In First Out*), ovvero tale per cui i dati vengono estratti in ordine rigorosamente inverso rispetto a quello in cui sono stati inseriti. Coda Per *coda*, invece, s'intende una Struttura Dati di tipo FIFO, *First In First Out* (il primo in ingresso è il primo ad uscire). Per implementare una coda viene utilizzato normalmente una Lista Concatenata. La Lista contiene due puntatori uno per la testa (il primo elemento della lista) e uno per la coda (ultimo elemento della Lista).

Coda [paragrafo 11.2.1 a pagina 45]

13 ESEMPI DI CODICI SORGENTI

Sito del Laboratorio d'Informatica I.

13.1 ESERCITAZIONE I

Vai al Testo dell'esercitazione

Codice 13.1: *fahrchels.c*

```c

/*
** Genera una tabella di conversione delle tempera-
** ture da Fahreneheit a Chelsius
**
** Tratto, con adattamenti, da Kernighan, Ritchie:
** The C programming language, Patience Hall PTR.
*/

#include <stdio.h>
#include <stdlib.h>

int main () {
    int lower, upper, step;
    float fahr, chelsius;

    lower = 0; /* Valore minimo, in gradi F,
                ** della tabella */
    upper = 300; /* Valore massimo, in grafi F,
                  ** della tabella */
    step = 20; /* Passo d'incremento della tabella */

    fahr = lower;
    while ( fahr <= upper ) {
        chelsius = (5.0/9.0) * (fahr - 32.0);
        printf ("%3.0f %6.1f\n", fahr, chelsius);
        fahr = fahr + step;
    }

    exit(EXIT_SUCCESS);
}
```

Codice 13.2: *valore_assoluto.c*

```c
/*
** Questo programma stampa il valore assoluto di un
** numero qualsiasi.
*/

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

int main (int argc, char * argv[]) {
    /* il numero che sara' inserito*/
```

```
    float  num;

    /* acquisisco i numeri */
    printf("Inserire un numero: ");
    scanf("%f", &num);

    /* stampo a schermo in valore assoluto */
    /* controlla che il numero sia positivo */
    if ( num >= 0 )
        printf("Valore assoluto: %f\n", num);
    else
        printf("Valore assoluto: %f\n", -num);

        exit (EXIT_SUCCESS);
}
```

13.2 ESERCITAZIONE II

Vai al Testo dell'esercitazione

Codice 13.3: *interi.c*

```
/*
** Questo programma legge dei numeri interi e ne
** stampa la somma, la differenza, il prodotto e
** il quoziente
*/

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

int main (int argc, char * argv[]) {
    int n1, n2; /* i numeri che saranno inseriti*/

    /* acquisico i numeri */
    printf("Inserire il primo numero: ");
    scanf("%d", &n1);
    printf("Inserire il secondo numero: ");
    scanf("%d", &n2);

    /* stampo a schermo le operazioni */
    printf("Somma: %d\n", n1+n2);
    printf("Differenza: %d\n", n1-n2);
    printf("Prodotto: %d\n", n1*n2);
    printf("Quoziente (intero): %d\n", n1/n2);

    exit (EXIT_SUCCESS);
}
```

Codice 13.4: *pari_dispari.c*

```
/*
** Questo programma stabilisce se un numero e' pari
** o dispari.
*/

# include <stdio.h>
# include <stdlib.h>
# include <math.h>
```

```
9
int main (int argc, char * argv[]) {
11      int n; /* il numero che sara' inserito */

13      /* acquisico il numero */
        printf("Inserire un numero: ");
15      scanf("%d", &n);

17      /* verifico che sia pari */
        if ( n%2 != 0)
19          printf("Il numero e' dispari\n");
        else
21          printf("Il numero e' pari\n");

23          exit(EXIT_SUCCESS);
}
```

13.3 ESERCITAZIONE III

Vai al Testo dell'esercitazione

Codice 13.5: *euro_dollaro.c*

```
/*
2 ** Questo programma stampa una tabella di conversio-
  ** ne da euro a dollaro tra valori arbitrari (scel-
4 ** ti a priori).
  */
6
  #include <stdio.h>
8 #include <stdlib.h>
  #include <math.h>
10


12 int main (void) {
      int i; /* dichiaro il contatore */
14    double dollaro, sterlina;

16    i = 1; /* primo valore del contatore */
      while ( i <= 20) {
18        dollaro = i*1.4424;
          sterlina = i*0.8823;
20        printf("%d euro - %lf dollari -  %lf sterli"
          "ne\n",  i, dollaro, sterlina);
22        i = i+1;
      }
24
      exit(EXIT_SUCCESS);
26 }
```

Codice 13.6: *primi.c*

```
/*
2 ** Questo programma stabilisce se un numero intero
  ** e' primo.
4 */

6 #include <stdio.h>
  #include <stdlib.h>
8 #include <math.h>
```

```
10 int main (void) {
       /* dichiaro i primi 10 numeri primi */
12     int p[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
       /* numero che verra' letto dal programma */
14     int num;
       /* variabile di controllo dello stato di "if" */
16     int success;
       int i, div; /* contatori */
18
       printf("Inserire un numero intero: ");
20     scanf("%d", &num);

22     success = 1; /* status del primo ciclo */
       i = 0;
24     while ( i < 10 ) {
           /* divide "num" per i numeri primi in p[] */
26         if ( num%p[i] == 0 && num != p[i] ) {
               printf("Il numero non e' primo\n");
28             /* cambia lo status del ciclo */
               success = 0;
30             break; /* esce dal ciclo */
           }
32         else
               i = i+1;
34     }

36     /* viene eseguito solo se lo status del primo
       ** ciclo e' diverso da 0 */
38     if ( success != 0 ) {
           /* divisore del numero inserito. Lo incre-
40         ** mento di 2 in modo che resti dispari */
           div = p[9]+2;
42         while ( div <= num/2 ) {
               if ( num%div == 0 ) {
44                 printf("Il numero non e' primo\n");
                   /* cambia lo status del ciclo */
46                 success = 0;
                   break /* esce dal ciclo */
48             }
               else
50                 div = div+2;
           }
52
           if ( success == 1 )
54             printf("Il numero e' primo\n");
       }
56
       exit(EXIT_SUCCESS);
58 }
```

Codice 13.7: *quadrati_cubi.c*

```
/*
** Questo programma legge venti numeri interi, li
** carica in un vettore, e successivamente calcola
** la somma dei quadrati e la somma dei cubi dei va-
** lori nel vettore, stampandoli in una tabella.
*/

#include <stdio.h>
#include <stdlib.h>
```

```c
#include <math.h>

int main (void) {
    int v[20]; /* vettore dei numeri inseriti */
    int i, e, f; /* contatori */
    int quad, cub; /* somme dei quadrati e dei cubi */

    i = 0;
    /* legge i 20 valori e li carica nel vettore */
    while ( i < 20 ) {
        printf("(%d) Inserire un numero intero: ",
        i+1);
        scanf("%d", &v[i]);
        i = i+1;
    }

    e = 1;
    /* calcola le somme quadratiche */
    quad = v[0]*v[0];
    while ( e < 20 ) {
        quad = quad + v[e]*v[e];
        e = e+1;
    }

    f = 1;
    /* calcola le somme dei cubi */
    cub = v[0]*v[0]*v[0];
    while ( f < 20 ) {
        cub = cub + v[f]*v[f]*v[f];
        f = f+1;
    }

    printf("Somma dei quadrati: %d || Somma dei cub"
    "i: %d \n", quad, cub);

    exit (EXIT_SUCCESS);
}
```

Codice 13.8: *somma_media_max.c*

```c
/*
** Questo programma legge dieci numeri interi, ne
** calcola la somma e la media, e stabilisce qual
**  e' il numero piu' grande.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main (void) {
    /* vettore, somma, massimo numero*/
    int p[10], somma, max;
    int h, i, f; /* contatori */

    h = 0;
    while ( h < 10 ) {
        printf("(%d) Inserire numero: ", h+1);
        scanf("%d", &p[h]);
        h = h+1;
    }
```

```
24      i = 0;
        somma = 0;
26      while ( i < 10 ) {
            somma = somma + p[i];
28          i = i+1;
        }
30      printf("\nSomma: %d\n", somma);
        printf("Media: %lf\n", (double) somma/10);
32
        f = 0;
34      while ( f < 10 ) {
            if ( p[f] < p[f+1])
36              max = p[f+1];
            else
38              max = p[f];
            f = f+1;
40      }
        printf("Il numero piu' grande e': %d\n", max);
42      exit(EXIT_SUCCESS);
}
```

13.4 ESERCITAZIONE IV

Vai al Testo dell'esercitazione

Codice 13.9: *esame.c*

```
1   /*
    ** Questo programma legge i voti riportati dai
3   ** cento studenti del corso di Laboratorio di
    ** Informatica I e determina la loro distribuzione.
5   ** Stabilisce, cioe' , quanti studenti hanno
    ** riportato il voto 1, quanti il voto 2, e cosi'
7   ** via, fino a 30.
    ** Il programma stabilisce anche quanti studenti
9   ** hanno superato l'esame.
    **
11  ** NOTA: per evitare di dover digitare cento numeri
    ** ad ogni esecuzione, procurarsi il file voti.txt,
13  ** copiarlo nella cartella che contiene il file
    ** eseguibile, ed eseguire il programma.
15  **
    ** Supponendo che l'eseguibile si chiami "esame",
17  ** dare:
    **
19  ** "./esame < voti.txt"
    */
21
    #include <stdio.h>
23  #include <stdlib.h>

25  int main (void)
    {
27      int i, v[100]; /* contatore, vettore voti */
        i = 0;
29      while ( i < 100 ) /* riempio il vettore */
        {
31          scanf("%d", &v[i]);
            i++;
```

```
    }
    /* contatore voti, vettore numero di voti */
    int h, n[30];
    h = 1;
    while ( h <= 30 )
    {
        int e;
        e = 0;
        /* gli array partono da n[0], non da n[1] */
        n[h-1] = 0;
        while ( e < 100 )
        {
            /* se nei voti esisete
            ** un numero uguale a h */
            if ( v[e] == h )
            {
                /* incrementa il numero di persone
                ** che ha preso quel voto */
                n[h-1]++;
            }
            e++;
        }
        printf("%d studenti hanno preso %d\n",
        n[h-1], h);
        h++;
    }

    /* contatore, numero di persone con voto > 17 */
    int f, pass;
    f = 18;
    pass = n[17];
    while ( f < 30 )
    {
        pass = pass + n[f];
        f++;
    }
    printf("\nAssumendo che il voto per cui l'esame"
    " si ritiene passato sia 18,\n%d studenti hanno "
    "passato l'esame\n", pass);

    exit(EXIT_SUCCESS);
}
```

Codice 13.10: *e_nepero.c*

```
/*
** Partendo dalla formula:
**
** e = Somme(0 -> + inf.)[1/k!]
**
** il programma calcola successive approssimazioni
**  del numero e.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* definisco la funzione "fattoriale" */
double fatt (int x) {
    if ( x == 0 || x == 1 )
        return 1;
```

```
    else {
        /* pongo "i = x-1" in modo che incrementando
        ** x non s'incrementi la i nel ciclo */
        int i = x-1;
        /* La condizione e' " i > 1 " perche'
        ** cosi' l'ultimo prodotto e' " x*2"
        ** visto che moltiplicare per 1 e' inutile
        */
        while ( i > 1 ) {
            x = x*i;
            i = i-1;
        }
        return x;
    }
}

/* funzione principale */
int main (void) {
    int n; /* passi della sommatoria */
    int e; /* contatore */
    double sum; /* valore della sommatoria */

    printf("Inserire il numero di passi della somma"
    "toria da eseguire: ");
    scanf("%d", &n);

    e = 0;
    sum = 0.0;
    while ( e <= n ) {
        sum = sum + 1.0/fatt(e);
        e = e+1;
    }

    printf("Valore (approssimato) di e: %lf\n", sum);
    exit(EXIT_SUCCESS);
}
```

Codice 13.11: *funzioni.c*

```
/*
** Questo programma calcola il valore di a^n per
** un numero "reale" a, e un intero n (anche negati-
** vo).Successivamente, calcola i valori delle fun-
** zioni:
**
** f(x) = 3x^3 - 2x^2 + (2x/5) - 1
** g(x) = x + (3/x^4)
**
** per valori di x in [-3, 3] a intervalli di am-
** piezza 0,25.
*/

#include <stdio.h>
#include <stdlib.h>

double pot (double a, int n) {
    if ( n == 0) /* a^0 = 1 */
        return 1;
    else if ( n > 0 ){
        int i;
        i = 1;
        while ( i < n ){
```

```
            a = a*a;
            i++;
        }
        return a;
    }
    else /*( n < 0 )*/{
        int i;
        i = 1;
        while ( i < n ) {
            a = (1/a)*(1/a);
            i++;
        }
        return a;
    }
}

int main (void) {
    /* calcolo i valori di f(x) con x in [-3, +3] */
    double x; /* contatre double (x) */
    x = -3.0;
    while ( x <= 3) {
        printf(" f(%lf) = %lf \n", x,
        3*pot(x, 3) - 2*pot(x, 2) + (0.4)*x - 1);
        x = x + 0.25;
    }

    /* calcolo i valori di g(z) con z in [-3, +3] */
    double z; /* contatre double (z) */
    z = -3.0;
    while ( z <= 3 ) {
        printf(" g(%lf) = %lf \n", z,
        z + (3/pot(z, 4)));
        z = z + 0.25;
    }

    exit(EXIT_SUCCESS);
}
```

Codice 13.12: *ordinamento.c*

```
/*
** Questo  programma  legge dodici numeri interi e
** li carica in un vettore. Successivamente, dispone
** i numeri in ordine crescente.
*/

#include <stdio.h>
#include <stdlib.h>

int main (void) {
    int i, p[13]; /* contatore, vettore */
    i = 0;
    while ( i < 12 ) {
        /* carico il vettore fino a p[11] */
        printf("%d) Inserire un intero: ", i+1);
        scanf("%d", &p[i]);
        i++;
    }

    int e, f; /* contatori */
    e = 0;
    while ( e < 11 ) {
```

```
        f = 0;
        while ( f < 11 ) {
            if ( p[f] > p[f+1] ) {
                /* uso p[12] come casella
                ** temporanea */
                p[12] = p[f];
                p[f] = p[f+1];
                p[f+1] = p[12];
            }
            f++;
        }
    e++;
    }

    int g;
    g = 0;
    while ( g < 12 ) {
        printf("%d\n", p[g]);
        g++;
    }

    exit(EXIT_SUCCESS);
}
```

13.5 ESERCITAZIONE V

Vai al Testo dell'esercitazione

Codice 13.13: *dadi.c*

```
/*
** Questo  programma simula una serie di 1000 lanci
** di due dadi. Esso stampare anche, alla fine della
** serie di lanci, la percentuale di uscita di ogni
** risultato (per risultato si intende la somma dei
** punteggi dei due dadi in ogni lancio).
**
** Alla fine, traccia un istogramma orizzontale del-
** le frequenze dei risultati.
*/

#include <stdio.h>
#include <stdlib.h>

int main (void) {
    int b, x;
    printf("Inserisci un intero per b: ");
    scanf("%d", &b);
    printf("Inserisci un intero per X(0): ");
    scanf("%d", &x);

    /* contatore ciclo, contatore numeri */
    int e, n[11];
    /* riempio il vettore di zeri */
    e = 0;
    while ( e < 12 ) {
        n[e] = 0;
        e++;
    }
```

```
31      /* produco i risultati dei dadi */
        int i = 1;
33      while ( i <= 10 ) {
            int g = 1;
35          while ( g <= 10) {
                int h=1;
37              while ( h <= 10) {
                    x = (b*x)%11 + 10;
39                  n[x-10] = n[x-10] + 1;
                    h++;
41              }
                b++; /* in modo da aumentare il fattore casuale */
43              g++;
            }
45          x++; /* in modo da aumentare il fattore casuale */
            i++;
47      }


49
        /* stampo la frequenza dei risultati trovati */
51      int f = 0;
        while ( f < 11 ) {
53          printf("La percentuale del risultato %d e':"
            "%lf per cento.\n", f+2, (double) n[f]/10);
55          f++;
        }
57
        /* stampo un istogramma */
59      int r;
        r = 0;
61      printf("\nISTOGRAMMA\n");
        printf("Numero : percentuale ( '-' = 1%)\n");
63      while ( r < 11 ) {
            printf("%d : ", r+2);
65          int t;
            t = 0;
67          while ( t < ((n[r]-2)/10) ) {
                printf("-");
69              t++;
            }
71          if ( t = (n[r]-2)/10 ) {
                printf("-|\n");
73              t++;
            }
75          r++;
        }
77      exit(EXIT_SUCCESS);
}
```

Codice 13.14: *distribuzione.c*

```
/*
2 ** Definiamo una successione di numeri interi x0,
  ** x1... con (x0 arbitrario) in questo modo (per
4 ** i > 0):
  **
6 ** xi = bx(i-1) (mod m)
  **
8 ** con b, m costanti. (mod m) denota il resto della
  ** divisione per m.
10 **
  ** Questo programma, fissati i valori di x0 , b e m,
```

```
/* produce i primi 2m valori della successione, te-
** nendo traccia della distribuzione dei valori gen-
** erati. Per ogni i compreso fra 0 e m, il program-
** ma conta quante volte i compare nella successione.
**
** Fissato m = 1024, cerca valori di b per cui la
** distribuzione sia uniforme, cioe' ogni valore di
** i compaia esattamente due volte.
*/

#include <stdio.h>
#include <stdlib.h>

/* calcolo i valori per cui la distribuzione e'
** uniforme */
int uni_search (int x) {
    int z;
    z = 1024;/* contatore ciclo, contatore numeri */
    int h, p[z];
    /* riempio il vettore di zeri */
    h = 0;
    while ( h < z ) {
        p[h] = 0;
        h++;
    }

    /* produco e stampo i termini della sommatoria */
    int j, r;
    j = 1;
    r = z + 1;
    while ( j <= 2*z ) {
        p[x] = p[x] + 1;
        x = (r*x)%z;
        if ( p[x] > 2 ) {
            r++;
            j = 1;
        }
        else
            j++;
    }
    return r;
}

int main (void) {
    /* raccolgo i dati */
    int m, b, x;

    printf("Inserisci un intero per m: ");
    scanf("%d", &m);
    printf("Inserisci un intero per b: ");
    scanf("%d", &b);
    printf("Inserisci un intero per X(0): ");
    scanf("%d", &x);

    /* contatore ciclo, contatore numeri */
    int e, n[m];
    /* riempio il vettore di zeri */
    e = 0;
    while ( e < m ) {
        n[e] = 0;
        e++;
    }
```

```
74
        /* produco e stampo i termini della sommatoria */
76      int i;
        i = 1;
78      while ( i < 2*m ) {
            n[x] = n[x] + 1;
80          x = (b*x)%m;
            printf("%d ", x);
82          i++;
        }
84
        if ( i == 2*m) {
86          n[x] = n[x] + 1;
            x = (b*x)%m;
88          printf("%d\n", x);
            i++;
90      }

92      /* stampo il n. di volte che un numero compare */
        int f;
94      f = 0;
        while ( f < m ) {
96          printf("Il valore %d e' comparso %d volte."
            "\n", f, n[f]);
98          f++;
        }
100
        printf("\nFissato m = 1024, il valore di b per "
102     "cui\nla successione e' uniforme: %d\n",
        uni_search(x));
104     exit(EXIT_SUCCESS);
    }
```

Codice 13.15: *schemaop_complessi.c*

```
1   /*
    ** Semplice messa in opera di numeri complessi con
3   **  operazioni aritmetiche
    */
5
    #include <stdio.h>
7   #include <stdlib.h>

9   #define N 2

11  /*
    ** La struttura n_complesso rappresenta un numero
13  ** complesso. Considera le possibili rappresenta-
    ** zioni alternative.
15  */
    struct n_complesso {
17      /* definisco un numero complesso
        ** in forma algebrica, anche se
19      ** potrei definirlo in termini di
        ** modulo e angolo
21      */

23      double reale; /* parte reale */
        double immag; /* parte immaginaria */
25  };

27  /*
```

```
** La riga che segue definisce un nome alternativo
** per il tipo 'struct n_complesso'.
*/
typedef struct n_complesso Cmpl;

/*
** La funzione 'somma' esegue la somma di due numeri
** complessi, ricevuti come parametri, costruisce un
** oggetto di tipo 'Cmpl' corrispondente alla somma,
** e lo restituisce.
*/
Cmpl somma (Cmpl a, Cmpl b) {
    Cmpl r;
    r.reale = a.reale + b.reale;
    r.immag = a.immag + b.immag;

    return r;
}

/*
** La funzione 'prodotto' esegue il prodotto di due
** numeri complessi, ricevuti come parametri, cos-
** truisce un oggetto di tipo 'Cmpl' corrispondente
** al prodotto, e lo restituisce.
*/
Cmpl prodotto (Cmpl a, Cmpl b) {
    Cmpl r;
    r.reale = a.reale*b.reale - a.immag*b.immag;
    r.immag = a.reale*b.immag + a.immag*b.reale;

    return r;
}

int main () {
    int i;
    Cmpl beta, gamma;
    Cmpl v[N];

    /*
    ** Fase di lettura dalla tastiera delle
    ** componenti di N numeri complessi.
    */
    for ( i=0; i < N; i++ ) {
        printf("\n%d) Inserire la parte reale: ",
        i+1);
        scanf("%lf", &v[i].reale);
        printf("%d) Inserire la parte immaginaria: ",
        i+1);
        scanf("%lf", &v[i].immag);
    }

    beta = somma(v[0], v[1]);
    gamma = prodotto(v[0], v[1]);

    printf ("\nSomma = [%lf, %lf]\n", beta.reale,
    beta.immag);
    printf ("Somma = %lf + i*%lf\n", beta.reale,
    beta.immag);
    printf ("\nProdotto = [%lf, %lf]\n",
    gamma.reale, gamma.immag);
    printf ("Prodotto = %lf + i*%lf\n", gamma.reale,
    gamma.immag);
```

```
91      exit(EXIT_SUCCESS);
}
```

13.6 ESERCITAZIONE VI

Vai al Testo dell'esercitazione

Codice 13.16: *calcolo_postfissa.c*

```
/*
** Per semplificare il programma, limitiamo le ope-
** razioni a somma (+) e prodotto (*). Limitiamo,
** inoltre i numeri ad interi con una sola cifra.
**
** In questo modo, non dobbiamo introdurre l'uso
** degli spazi per identificare univocamente i nu-
** meri. Ulteriore vincolo e' che si suppone che
** l'espressione introdotta non sia affetta da er-
** rori.
*/

#include <stdio.h>
#include <stdlib.h>
/* contiene la funzione "isdigit()" */
#include <ctype.h>

#define N 50

struct pila {
    /*
    ** per tenere insieme due variabili
    ** non omogenee, usiamo un record
    */
    int val[N];
    int indice;
} valori;

void push (int v) {
    /*
    ** consideriamo uno stack come fissato, e suppo-
    ** niamo che la funzione non produca risultati
    **
    */
    valori.val[valori.indice] = v;
    valori.indice++;
    /*
    ** questa funzione non da' errore finche' il
    ** vettore non e' pieno. Se valori.indice = 50,
    **  c'e' un errore.
    */
}

int pop (void) {
    valori.indice--;
    return valori.val[valori.indice];
    /*
    ** questa funzione da' errore se il vettore e'
    ** vuoto. Se valori.indice = 0, c'e' un errore.
    */
```

```
}

int val (void) {
    /*
    ** legge le sequenze di caratteri e compie
    ** operazioni sui caratteri immessi
    */
    int x, y;
    char c;
    printf ("Inserisci l'espressione: ");
    c = getchar();
    while ( c != '\n') {
        /* isdigit(c) produce 1 se c non e' una
        ** cifra, 0 altrimenti */
        if ( isdigit(c) )
            push(c - '0');
        else if ( c == '+') {
            x = pop();
            y = pop();
            push(x+y);
        }
        else /* if ( c == '*') */ {
            x = pop();
            y = pop();
            push(x*y);
        }
        c = getchar();
    }
    return pop();
}

int main (void) {
    valori.indice = 0;
    printf("Risultato: %d\n", val());
    exit(EXIT_SUCCESS);
}
```

Codice 13.17: *infix_postfix.c*

```
/*
** Questo programma traduce un'espressione algebrica
** dalla notazione infissa a quella postfissa. Fun-
** ziona solo per numeri di una sola cifra. Affin-
** che' funzioni, bisogna includere ogni espressione
** tra parentesi tonde.
*/

#include <stdio.h>
#include <stdlib.h>
/* contiene la funzione "isdigit()" */
#include <ctype.h>

#define N 50
/* stack */
struct pila {
    char val[N];
    int indice;
} valori;

void push (char v) {
    /*
    ** consideriamo uno stack come fissato, e sup-
```

```
    ** poniamo che la funzione non produca risultati
    */
    valori.val[valori.indice] = v;
    valori.indice++;
    /*
    ** questa funzione non da' errore finche' il
    ** vettorenon e' pieno. Se valori.indice = 50,
    ** c'e' un errore.
    */
}

/* non necessita di argomenti */
char pop (void) {
    valori.indice--;
    return valori.val[valori.indice];
        /*
        ** questa funzione da' errore se il vettore
        ** e' vuoto. Se valori.indice = 0, c'e' un
        ** errore.
        */
}

char traduzione (void) {
    char c, temp;
    printf ("Inserisci notazione infissa (usa le pa"
    "rentesi): ");
    c = getchar();
    while ( c != '\n') {
        /* isdigit(c) produce 1 se c non e' una
        ** cifra, 0 altrimenti.
        ** E' un valore booleano, per questo la con-
        ** dizione puo' essere scritta cosi' */
        if ( isdigit(c) ) {
            printf("%c", c);
        }
        else if ( c == '(' || c == '+' || c == '*' ) {
            push(c);
        }
        else if ( c == ')' ) {
            printf("%c", pop() );
            temp = pop();
        }
        c = getchar();
    }
    printf("\n");
}

int main (int argc, char *argv[]) {
    /* punta inizialmente alla prima posizione
    del vettore */
    valori.indice = 0;
    traduzione();
    exit(EXIT_SUCCESS);
}
```

Codice 13.18: *radice.c*

```
/*
** La radice quadrata di un numero reale puo' essere
** calcolata, in modo approssimato, come segue: sia
** x il numero del quale vogliamo calcolare la ra-
** dice quadrata; stabiliamo una soglia 's' relativa
```

```c
** alla precisione del risultato; partiamo da una
** stima g; se |g^2 - x| < s, consideriamo g come
** risultato accettato; altrimenti calcoliamo una
** nuova stima per mezzo della formula (x/g + g)/2 e
** ripetiamo il procedimento.
**
** Questo un programma che applica la procedura des-
** critta e stampa il numero di iterazioni compiute
** per arrivare al risultato.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* funzione "valore assoluto */
double v_assoluto (double n) {
    if ( n >= 0)
        return n;
    else
        return -n;
}

double stima (double x, double g) {
    if ( g > 0 && g < x) {
        g = ((x/g)+g)/2;
        return g;
    }
    else
        return -1;
}

/* funzione "radice quadrata" */
double radice (double x, double s) {
    double g;

    if ( x < 0)
        g = -1;

    else if ( x == 0)
        g = 0;

    else if ( x == 1 )
        g = 1;

    else  {
        if ( x > 0 && x < 1)
            g = x;
        else if (x > 1 && x < 2)
            g = 1;
        else if ( x >= 2)
            g = x/2;

        while ( v_assoluto(pow(g, 2)-x) >= s ) {
            g = stima(x, g);
            /* se ci sono errori in "stima(x, g)",
            ** g = -1 */
            if (stima(x, g) == -1)
                break;
        }
    }
    return g;
```

```
}

int main (void) {
    double y, p, r;
    printf("Inserire il radicando: ");
    scanf("%lf", &y);
    printf("Inserire il grado di precisione: ");
    scanf("%lf", &p);

    r = radice(y, p);
    if ( r != -1) {
        printf("\nRadice approssimata di %lf: %lf"
        "\n", y, r);
        exit(EXIT_SUCCESS);
    }
    else
        printf("ERRORE");
        exit(EXIT_FAILURE);
}
```

13.7 ESERCITAZIONE VII

Vai al Testo dell'esercitazione

Codice 13.19: *binario_decimale.c*

```
/*
** Nella rappresentazione in base 2 di un numero in
** un vettore, per stabilire quale sia l'ultima ci-
** fra, collochiamo il valore '-1' nella posizione a
** destra dell'ultima cifra significativa. Ad esem-
** pio: 181 = 128 + 32 + 16 + 4 + 1 corrisponde ad
** un vettore contenente, nell'ordine, i valori
** 1 0 1 1 0 1 0 1 -1.
**
** Questo programma contiene due funzioni, decbin e
** bindec. La prima riceve come argomento un numero
** intero n e un vettore, e scrive nel vettore la
** rappresentazione binaria di n. La seconda, par-
** tendo da un vettore contenente le cifre della
** rappresentazione binaria di un numero, resti-
** tuisce il numero stesso.
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define N 50

/* struttura contenente un binario */
struct binario {
    int v[N];
    int indice;
} bin_num;

/* funzione push */
void push (int d) {
    bin_num.v[bin_num.indice] = d;
```

```
    bin_num.indice++;
}

/* converte da decimale a binario */
int decbin (int n) {
    int success; /* variabile di controllo */
    success = 1; /* variabile di controllo vera */
    push(-1);
    while (n/2 != 0) {
        /* scrive il resto nel vettore binario */
        push(n%2);
        n = n/2; /* aggiorna n */
        /* se il vettore finisce */
        if ( bin_num.indice > N-2 ) {
            /* variabile di controllo falsa */
            success = 0;
            /* interrompe il ciclo */
            break;
        }
    }
    /* se il quoziente e' 0 */
    if ( n/2 == 0 && success == 1 )
        push(n%2);
    return success;
}

int bindec ( int c ) {
    int i, z[N];
    /* c e' una cifra e i non supera la lunghezza
    ** del vettore */
    for ( i = 0; isdigit(c) && i < N-2; i++) {
        z[i] = c -'0'; /* assegno  z[i] = 'c'-'0' */
        c = getchar();
    }

    int e, sum;
    sum = 0;
    for ( e = i-1; e >= 0; e-- )
        sum = sum + z[e] * pow(2, (i-1) - e);
    return sum;
}

int main (int argc, char *argv[]) {
    int x, c;
    printf("Inserire un numero in base 2: ");
    c = getchar();
    x = bindec(c);
    printf("Rappresentazione decimale: %d \n", x );
    /* la prima posizione vuota del vettore binario
    ** e' 0 */
    bin_num.indice = 0;
    int m;
    printf("\nInserire un numero in base 10: ");
    scanf("%d", &m );

    printf("Rappresentazione binaria: ");
    /* se non ci sono stati errori */
    if ( decbin(m) != 0 ) {
        int e;
        for( e = bin_num.indice - 1; e >= 1; e--)
            /* stampa le cifre binarie */
            printf("%d", bin_num.v[e]);
```

```
        if ( bin_num.v[e] == -1 )
98          /* va a capo alla fine */
            printf("\n");
100     }
        else /* if ( decbin(m) == 0) */
102         printf("Numero troppo grande\n");

104     exit(EXIT_SUCCESS);
}
```

A APPUNTI VARI

A.1 STACK

Come già detto, si può usare un vettore come *stack* (pila), dichiarandolo abbastanza lungo e supponendo che le operazioni aritmetiche stiano all'interno della lunghezza del vettore. Ci si può servire di una variabile intera che punti alla prima posizione libera del vettore. Per ogni operazione *push*, si copia il nuovo valore nella posizione dell'indice e s'aggiorna il valore di quest'ultimo. Nel caso dell'operazione *pop* si dovrà leggere il valore della variabile due caselle a sinistra dell'indice e spostare quest'ultimo di una posizione a sinistra.

A.2 EDITOR DI TESTO VI(M)[1]

Sull'editor di testo *vi*, ci sono due modalità:

- Modo inserimento;
- Modo comando.

Quando il programma viene avviato, è in *modalità comando*, in attesa di ricevere un particolare comando. Premendo il tasto "I", l'editor passa in *modalità inserimento*. Premendo il tasto "A" in modalità comando, si comincia a scrivere a destra del cursore. Per uscire da questa modalità: "Esc". Il tasto "O" apre una nuova riga sotto la posizione del cursore, mentre la combinazione "Maiusc+O" ne apre una sopra la posizione del cursore. Quando il cursore si sposta su una parentesi aperta, il programma evidenzia la parentesi chiusa corrispondente. Per cancellare un carattere, in modo comando, basta spostare il cursore sopra il carattere da cancellare e premere il tasto "X". Per cancellare l'intera riga, ci si sposta all'inizio della riga e si preme il tasto "D" (2 volte). In modalità comando, alla pressione del tasto ":" (Maiusc+.) il programma aspetta si dei comandi, ad esemprio, per salvare ":w" (*write*), per uscire e ritornare all'*ambiente shell* ":q" (*quit*). A questo punto, volendo, è possibile tornare a lavorare sul file salvato[2].

A.3 DEBUGGER

Il *debugger* è un programma progettato per l'analisi e l'eliminazione dei bug (vedi il paragrafo A.4 nella pagina successiva) presenti in altri software. Assieme al compilatore è fra i più importanti strumenti di sviluppo a disposizione di un programmatore. Il compito principale del debugger è di mostrare il frammento di codice che genera il problema (tipicamente un *crash*).

1 la versione corrente è *vi improoved*.
2 Il comando ":w" può salvare il contenuto del file non nel file in cui si sta lavorando, ma in uno diverso (in pratica lo copia) scrivendo ":w *nomefile*". ":wq" salva ed esce.

A.4 BUG

Nell'informatica il termine *bug* (o baco) identifica un errore nella scrittura di un programma software. Meno comunemente, può indicare un difetto di progettazione in un componente hardware. Un bug di un programma è un errore che porta al malfunzionamento dello stesso. La causa del maggior numero di bug è spesso il codice sorgente scritto da un programmatore, ma può anche accadere che venga prodotto dal compilatore.

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI CODICI

ELENCO DEGLI ACRONIMI

OS Operating System

ACM Association For Computing Machinery

NPL National Physical Laboratory

ACE Automatic Computing Engine

MADAM Manchester Automatical Digital Machine

BIBLIOGRAFIA

[1] Sito del prof. luca bernardinello, 2011. URL http://www.mc3.disco.unimib.it/lif/.

[2] Wikipedia (en), 2011. URL http://en.wikipedia.org/wiki/Main_Page.

[3] Wikipedia (it), 2011. URL http://it.wikipedia.org/wiki/Pagina_principale.

[4] Il problema dello zaino su wikipedia (it), 2011. URL http://it.wikipedia.org/wiki/Problema_dello_zaino.

[5] L.M. Barone, E. Marinari, G. Organtini, and F. Ricci-Tersenghi. *Programmazione scientifica*. Pearson Education, 2006.

[6] J.G. Brookshear. *Informatica - Una panoramica generale*. Pearson-Addison Wesley, 2004.

[7] Marcello Frixione and Dario Palladino. *La computabilità: algoritmi, logica, calcolatori*. Le Bussole. Carocci, 2011.

[8] Daniele Giacomini. Appunti d'informatica libera, 2011. URL http://a2.pluto.it/a2.htm.

[9] Antonio Giorgilli. Note su linguaggio c e dintorni.

[10] B.W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[11] B.W. Kernighan and D.M. Ritchie. *Il linguaggio C*. Pearson-Prentice Hall, 2004. Seconda edizione.

[12] Alfio Quarteroni and Fausto Saleri. *Calcolo Scientifico*. Springer, 2008. Quarta edizione.

[13] G.M. Schneider and J.L. Gersting. *Informatica*. Apogeo, 2007.